Parallel Architectures for Entropy Coding in a Dual-Standard Ultra-HD Video Encoder

by

Bonnie K. Y. Lam

B.A.Sc. in Engineering Physics
University of British Columbia, 2008

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Author .....
Department of Electrical Engineering and Computer Science
May 21 2010

Certified by ....
Anantha P. Chandrakasan
Joseph F. and Nancy P. Keithley Professor of Electrical Engineering
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chair, Department Committee on Graduate Students

Parallel Architectures for Entropy Coding in a Dual-Standard Ultra-HD Video Encoder

by

Bonnie K. Y. Lam

Submitted to the Department of Electrical Engineering and Computer Science
on May 21 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

The mismatch between the rapid increase in resolution requirements and the slower increase in energy capacity demand more aggressive low-power circuit design techniques to maintain battery life of hand-held multimedia devices. As the operating voltage is lowered to reduce power consumption, the maximum operating frequency of the system must also decrease while the performance requirements remain constant. To meet these performance constraints imposed by the high resolution and complex functionality of video processing systems, novel techniques for increasing throughput are explored. In particular, the entropy coding functional block faces the most stringent requirements to deliver the necessary throughput due to its highly serial nature, especially to sustain real-time encoding. This thesis proposes parallel architectures for high-performance entropy coding for high-resolution, dual-standard video encoding.

To demonstrate the most aggressive techniques for achieving standard reconfigurability, two markedly different video compression standards (H.264/AVC and VC-1) are supported. Specifically, the entropy coder must process data generated from a quad full-HD (4096x2160 pixels per frame, the equivalent of four full-HD frames) video at a frame rate of 30 frames per second and perform lossless compression to generate an output bitstream. This block will be integrated into a dual-standard video encoder chip targeted for operation at 0.6V, which will be fabricated following the completion of this thesis.

Parallelism, as well as other techniques applied at the syntax element or bit level, are used to achieve the overall throughput requirements. Three frames of video data are processed in parallel at the system level, and varying degrees of parallelism are employed within the entropy coding block for each standard. The VC-1 entropy encoder block encodes 735M symbols per second with a gate count of 136.6K and power consumption of 304.5 μW, and the H.264 block encodes 4.97G binary symbols per second through three-frame parallelism and a 6-bin cascaded pipelining architecture with a critical path delay of 20.05 ns.

Thesis Supervisor: Anantha P. Chandrakasan
Title: Joseph F. and Nancy P. Keithley Professor of Electrical Engineering

Acknowledgments

The amount of growth I have experienced since the first day I stepped foot on MIT campus is remarkable, and I often look back in amazement at how much I have matured not only as a researcher but as an individual as well. This thesis marks the conclusion of an adventurous and fulfilling experience that would not have been possible without the support of these wonderful people.

I owe my deepest gratitude to my thesis advisor, Professor Chandrakasan, for his constant inspiration and support. From the day I received his warm welcome to MIT that changed my life forever, he has continued to show me the art of research through his leadership and example. Not a single meeting with him goes by without my marveling at the amount of attention and interest he shows to each of his many students, and the level of technical fluency he maintains in the wide range of work being done in his research group. He has the presence of a skilled manager and mentor, but shares his sense of humour as generously as he shares his wisdom. I am certain that I have much to learn from him as I embark on my Ph.D. under his guidance, and I look forward to developing a fruitful relationship with him.

My family has been the first and most eager to support me in my decisions, and even though we are physically apart we remain close at heart. Mom, you have always been the first to believe in me even when I did not believe in myself. Thank you for teaching me your strength as an individual, and for trusting that the extra distance and flight connection to Boston is well worth the trip! Dad, you are always so outspoken about the pride and faith you have for me. Even though I blush out of modesty when you say these things, your words of encouragement have really pulled me. Debbie, I have learned so much more from you that I can ever imagine teaching you as your big sister. You are a better childhood companion than I could ever ask for – thanks for switching roles with me all the time!

I would like to thank Mario, my best friend and colleague, for his continued support both emotionally and professionally and for never failing to make me laugh. He has taught me a great deal about how to be a good engineer and a good friend by example, and encouraged me to reach beyond my limits.

I have also learned a great deal from my colleagues in the Ananthagroup. Special thanks to Vivienne and Chih-Chi are in order for being wonderful mentors and always ready to

lend a helpful hand with research. Rahul and Mahmut, it has been great working with you. Yildiz, you have been a great friend and I really enjoyed all our deep conversations. To everyone else in the group, thanks for being such a welcoming and supportive crowd – the random fun conversations around lab have certainly made research at MIT much more enjoyable. To Margaret, thanks for always looking out for us and I hope we both find the lab to be at a comfortable temperature in the future.

To my friends at MIT, thank you for being such a wonderful crowd. To Karen, thanks for being a great friend and introducing me to almost half the people I know at MIT. To Grace, it's been only a year or so but our friendship has grown so fast! To Audrey, thanks for being a great listener and a great cook! To Lei, Ying, and Ermin, my Karate buddies, thanks for contributing to the betterment of my physical and mental health, especially in the past year. To everyone else, it's been so much fun hanging out with you and I hope our friendships continue to grow.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

As high-performance consumer electronics, in particular portable devices such as camera phones, digital-still cameras, and high-resolution video recorders, become increasingly popular, the amount of power consumed by these electronics to deliver these stringent performance requirements has also increased. The batteries providing this power must necessarily scale up in size, but this trend is opposed by the market demands for smaller and lighter devices. In order to circumvent the need to increase battery size and consequently form factors of these products, low-power circuit design techniques must be employed to reduce the amount of power needed for a given performance level. Only by reducing the power consumption of integrated circuits on these devices can one hope to meet the opposing demand for smaller consumer electronics.

Power consumption in digital CMOS circuits can be quantified by the following relationship:

$$\begin{aligned} P_{total} &= P_{switching} + P_{shortcircuit} + P_{leakage} \\ &= p_t \times (C_L \times V_{DD}^2 \times f_{clk}) + I_{sc} \times V_{DD} + I_{leakage} \times V_{DD} \end{aligned} \tag{1.1}$$

The contribution of short circuit power is negligible compared to the other two components in static CMOS circuits. Reduction in V_{DD} provides an even greater degree of

dynamic power reduction than reduction in load capacitance C_L due to the quadratic dependency of $P_{switching}$ on V_{DD}, and leakage power will also decrease with reduced supply voltage. However, the benefits of voltage scaling cannot be exploited constraint-free, due to the inverse relationship between delay and supply voltage, as shown here [2]:

$$T_{delay} = \frac{C_L \times V_{DD}}{I} = \frac{C_L \times V_{DD}}{\mu C_{ox}(W/L)(V_{DD} - V_t)^{\alpha}} \tag{1.2}$$

where $1 < \alpha \leq 2$. Thus, decreasing V_{DD} effectively increases the delay in the circuit. Since the operating frequency is bound by the maximum critical path delay in the circuit, maximum operating frequencies are reduced as V_{DD} reduces. To meet the same throughput requirements imposed on circuits operating at nominal voltage, parallel architectures are needed in low-voltage designs.

One prominent example of an application that warrants power considerations is video encoding and decoding. In addition to the traditional hand-held video recorders and video players that specifically target this application, smartphones that feature PC-like functionality are emerging as major players in the consumer electronics market, and these devices require even more processing power per unit weight. Since video recording and playback are key features on the typical smartphone, the design of low-power video encoder and decoder chips presents itself as an important research problem.

A typical video encoder is comprised of several major blocks performing the following operations: motion estimation, transforms, intraprediction, and entropy coding. A block diagram illustrating these functional blocks is shown in Figure 1-1. The entropy coding unit converts the video data encoded by the other blocks into a bit sequence representation, known as a bitstream. This thesis presents the design and implementation of a high-performance entropy encoding unit supporting two markedly different video compression standards (H.264/AVC and VC-1). Specifically, the entropy coder must process data generated from an ultra high definition or ultra-HD (4096x2160 pixels per frame) video at a frame rate of 30 frames per second and perform lossless compression to generate an output bitstream. In applications such as video conferencing and video recording where real-time video compression is required, this resolution and frame rate specification places stringent

requirements on the throughput of the entropy encoder. This block will be integrated into a dual-standard video encoder chip targeted for low-voltage operation at 0.6V, which will be fabricated following the completion of this thesis.

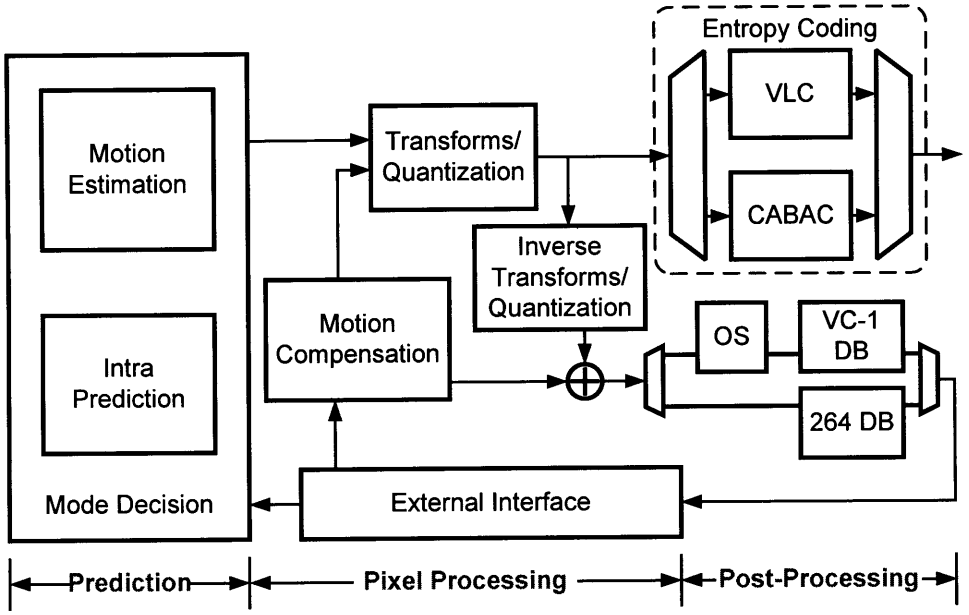

Figure 1-1: Functional block diagram of a video encoder system (*Courtesy of Dr. Chih-Chi Cheng*).

1.2 Entropy Coding in VC-1 and H.264

Entropy coding describes the set of lossless data compression techniques by which the compressed data can be used to exactly reconstruct the original data. The two most common entropy coding schemes are variable length (or Huffman) coding and arithmetic coding. Both of these algorithms generate a bitstream that represents the encoded information. Each of the entropy coding techniques reduces the number of bits needed to represent the source information by assigning a unique representation to each possible value of the source data, thus compressing the data. Individually, each codeword may be shorter or longer than the original symbol (also referred to as the source symbol) itself, but on average fewer bits are used in the compressed output than would have been needed if the data were used directly.

In the context of video encoding, entropy coding is the functional block that generates

the sequence of bits used in video transmission and storage. This sequence is known as the bitstream, and is comprised of the consecutive segments of bits. Each of these segments represents a unit (or element) of video data, referred to as a syntax element. The input and output of an entropy encoder is illustrated in Figure 1-2. The method by which these bits are generated is generally referred to the entropy coding method. Depending on the specifications defined in a particular video standard, one or more entropy coding methods can be used to generate the bitstream representation of video sequence.

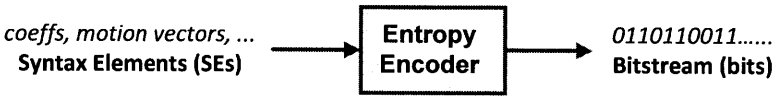

Figure 1-2: Input and output of an entropy encoder.

The SMPTE 421M video codec standard, informally known as VC-1, uses a multi-table variable length coding scheme [3]. The video data, represented by the syntax elements supplied to the entropy coding block from all other blocks in the encoder system, are mapped to the bitstream through many table lookups. The concept of variable length coding and techniques that have been explored for efficient implementation are discussed in Section 1.3.

The H.264 video standard, on the other hand, employs the more complex context-based adaptive binary arithmetic coding (CABAC) scheme [1]. By representing a sequence of binary symbols of arbitrary length using a single fractional number, this method can yield a higher amount of compression than other entropy coding schemes, thus justifying the increased algorithmic complexity. The mechanisms employed in CABAC and previous works discussing various implementations, particularly those in hardware, are detailed in Section 1.4.

1.3 Variable-Length Coding

1.3.1 Concept

Variable length coding, also known as Huffman coding, was developed by David A. Huffman at MIT and published in 1952 in [4]. This method uses a variable length code (VLC) table

to map each possible value of the source symbol to a unique codeword. The table is derived based on the likelihood of occurrence of each value and assigns the shortest codeword to the most likely value this ensures that the fewest number of bits are used in most of the bit sequence generated. The likelihood of occurrence is essentially the estimated probability of the source symbol to have a certain value. The set of all probability values is known as the probability model. The quality of this model, in other words how closely the estimated probabilities match the true data statistics, affects the amount of compression obtained. For instance, in the extreme case that the most common value of the source symbol was assigned the lowest probability when generating the code table, a very long codeword would appear many times in the output bitstream, and the total length of the bitstream would dramatically increase. Figure 1-3 depicts the procedure by which a variable length code table is generated.

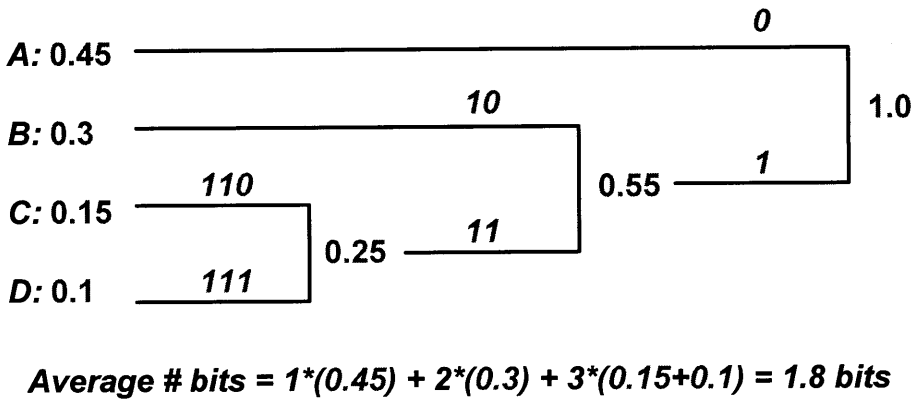

Figure 1-3: Huffman tree example.

All possible values of the source symbol to be encoded are listed in order of decreasing probability (probabilities are shown in red). These form the leaf nodes of the final binary tree. First, an internal node is created by summing the probabilities of the two least probable values, i.e. the bottom two leaf nodes. This new internal node then replaces the two leaf nodes in the set. The same process is repeated on the set until a root node is created, with probability 1. For each parent node in the tree, the more probable child node is assigned bit 0 and the other bit 1. The codeword for each leaf node is then read from the root to leaf, as indicated in blue italics.

1.3.2 Previous Work

While VC-1 is a relatively new video coding standard and as such there is little published work specifically addressing entropy coding for VC-1, the topic of variable length coding and efficient hardware implementations of both encoder and decoders have been investigated extensively in the context of other application spaces and other more established video standards. A variable length coder with barrel-shift-based concatenation targeting digital high-definition television (HDTV) applications is presented in [5]. This implementation features single-cycle-encoding for each codeword regardless of its length. An entropy decoder supporting multiple video standards including VC-1 and H.264 using a generic table partitioning strategy is proposed in [6]. Efficient architectures for implementing multiple modes of bitplane coding in parallel are discussed in [7]; however, since only one mode allows each bit of information to be encoded as it becomes available real-time, the parallel architectures are not applicable to our design.

The variable length decoder algorithm and architecture proposed in [8] is designed for MPEG-2 entropy decoding. A test chip was fabricated and post-layout simulations showed partitioning large codeword lookup tables and reducing the width of the barrel shifter results in a significant amount of area and power savings. Although these techniques were demonstrated on the decoding process, some of the fundamental concepts can be applied to the encoding process as well – these are investigated, implemented, and discussed in Chapter 3.

1.4 CABAC

1.4.1 Concept

Arithmetic coding, as described in [9,10], is a lossless data compression scheme that converts an entire message into a single number, as opposed to Huffman coding which represents each symbol with an individual codeword. This method also assigns fewer bits to more probable symbols and more bits to less probable symbols, reducing the overall number of bits used. Arithmetic coding provides greater compression since it is possible to represent one bit of data using a fractional number of bits, whereas Huffman coding can at best achieve the minimum integral number of bits, in other words a single bit.

To encode a sequence of symbols, the range between 0.0 and 1.0 is divided into n

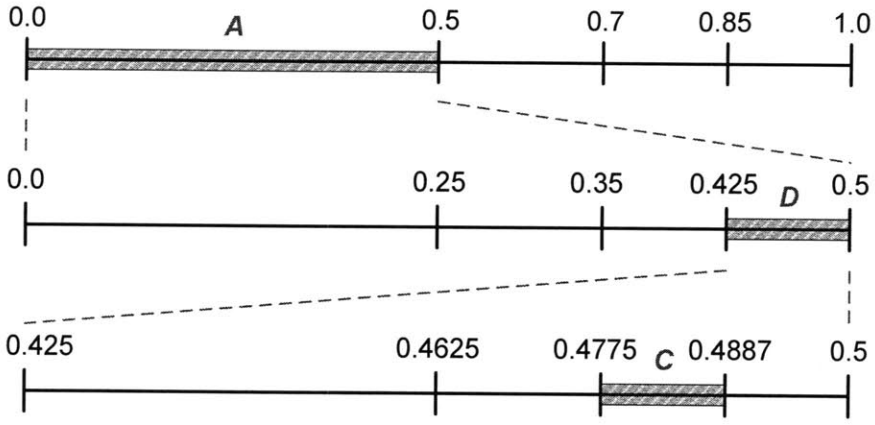

Figure 1-4: Encoding "ADC" using Static Arithmetic Coding.

intervals, one for each possible value of the symbol. The length of each interval is equal to the probability of the corresponding value, thus the n intervals have a total length of 1.0. The value of the first symbol determines which of these intervals becomes the next interval. The same division into sub-intervals is recursively performed on this interval, and the next symbol to be encoded selects one of the intervals as the new current interval. The final interval is determined after m such iterations, where m is equal to the number of symbols in the sequence. The minimum number of bits required to represent all fractions within that interval is taken as the final number to be transmitted. This number is a fraction that lies between 0.0 and 1.0. Figure 1-4 illustrates an example of the procedure described above, with n=4 (possible symbol values A, B, C, and D with respective probabilities 50%, 20%, 15%, and 15%) and m=3 (a sequence of length 3) a point within the final range (between 0.4775 and 0.48875) unique represents the sequence ADC given the proper probability model.

In particular, context-based adaptive binary arithmetic coding (commonly referred to as *CABAC*) is becoming increasingly popular due to its higher coding efficiency. This scheme is a variation of the binary arithmetic coding method, which is a special case of the arithmetic coding described above (with n=2). Given only two possible values for a symbol, the recursion described above is performed to determine the final interval representing the message, as shown for the example sequence "ABA" in Figure 1-5. In this example the symbol is assumed to take the value "A" with a probability of 70% and the value "B" with a probability of 30%. One advantage of limiting the symbols to be encoded to be binary

is that it simplifies the bookkeeping required - the probability of occurrence of one of the two values fully characterizes the subdivision at each stage. The interval selected at each stage can be represented by the lower bound and range (or length) of the interval. Symbols that are not binary can be passed through a pre-processing block that generates a binary representation using other entropy coding methods such as unary and exponential-Golomb coding before being encoded using binary arithmetic coding.

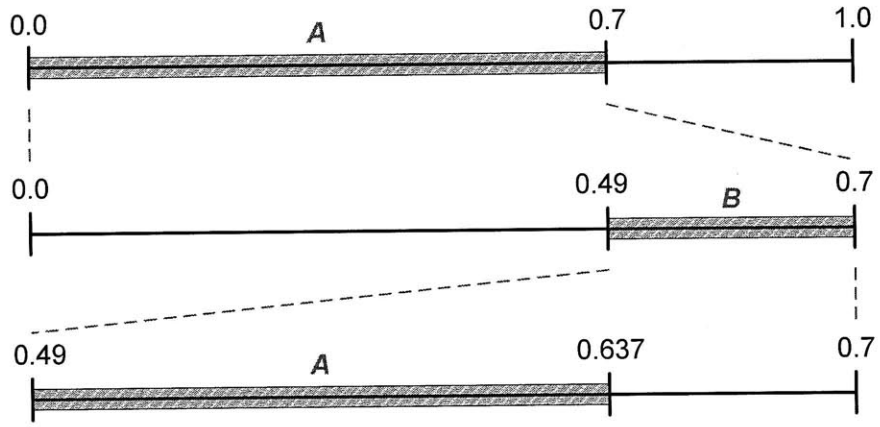

Figure 1-5: Encoding "ABA" using Static Binary Arithmetic Coding.

The "Context-based Adaptive" portion of CABAC refers to updating the probability model (known as the context) between stages of encoding. This can happen in two scenarios:

1. As the known values of each additional symbol is encoded, the probability model is updated to reflect the data statistics;

2. or, the probability model to be used changes (resulting in a switching of contexts) because the new symbol does not use the same context as the last symbol encoded.

An example of encoding the sequence "ABA" with changing probability values (due to one of the two conditions above) is shown in Figure 1-6.

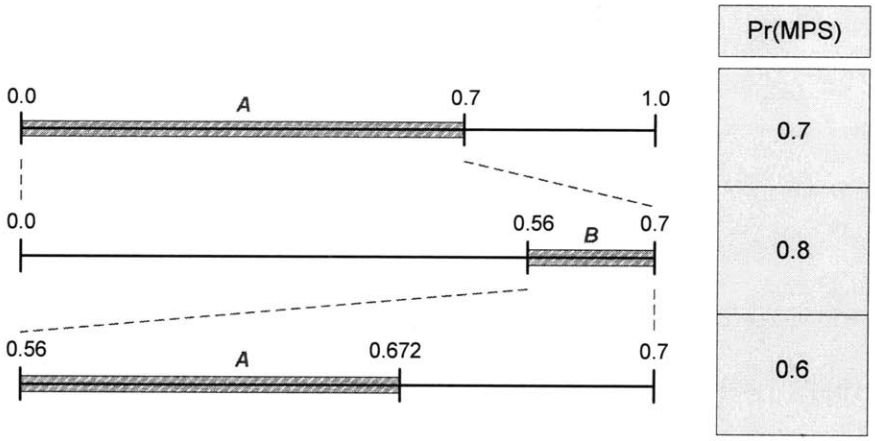

Figure 1-6: Encoding "ABA" using Context Adaptive Binary Arithmetic Coding (CABAC).

1.4.2 Previous Work

Work in [11] selects a different dataflow based on the pattern of consecutive bins to allow processing more than one symbol per cycle. However, this approach is highly data-dependent and does not improve average performance significantly, since it only succeeds to encode multiple symbols in some specific cases. Authors of [12] propose the use of task rescheduling, algorithm-specific early fetch, pipeline bypass, and bubble insertion in a pipelined architecture. Use of a variable bit length tag cache and a register file is presented in [13]. Encoding two binary symbols per cycle, the dual-standard encoder supporting JPEG2000 and H.264 proposed in [14] uses pre-computation to encode more than one binary symbol per cycle, but an accurate throughput is not reported.

A pipeline architecture is revisited in [15], but data dependencies between consecutively encoded symbols are handled by stalling the pipeline and thus have a negative impact on throughput. In [16], random-access memories are used to store the information from neighbouring blocks, but this access again causes pipeline stalls. A cascaded structure is used in [17] to process more than one binary symbol per cycle, but since the overall critical path is increased, the overall throughput (measured in binary symbols processed per second) suffers from the lower clock frequency. Multi-bin encoding (where multiple binary symbols are encoded per cycle) is explored in [18] through the use of multiple SRAM banks to provide sufficient parallel ports for data access. CABAC encoding with rate-distortion optimization (RDO) is investigated in [19], and throughput is mostly limited by the single context access

available. A multi-stage pipeline yielding a throughput of 1 bin per cycle is first proposed in [20], and further work demonstrating 2 bins/cycle using a pre-fetch scheme with a small cache is presented in [21]. Rate distortion optimization is implemented and context memory accesses are done in a row-based fashion to reduce accesses in [22].

1.5 System Level Specifications for Dual-Standard Video Encoder

Many of the published works have addressed the design challenges for a single video standard and/or a single entropy coding method. In particular, the CABAC scheme has garnered much attention due to its design challenges in its high complexity and serial nature. However, the video encoder system design that encompasses this work aims to support high-resolution video coding for two widely adopted video compression standards, namely H.264 and VC-1, the former supporting CABAC entropy coding and the latter supporting VLC entropy coding. To support these two fundamentally different entropy coding schemes, two separate entropy encoders must be designed with consideration given to intelligent resource sharing techniques and common timing constraints to achieve low power encoding while meeting system level throughput requirements imposed by both standards.

The quad full high definition resolution sets a stringent design requirement on the throughput of the video encoder system due to the quadruple increase in number of pixels per video frame as compared to the full-HD format known as 1080p. With the goal of low-power operation in mind, this means that highly parallel architectures are necessary. The video encoder chip employs frame and macroblock parallelism, where three frames and two macroblocks from each frame are processed in parallel. This level of parallelism ensures that the required throughput can be met despite the lowered clock speed directly resulting from low-voltage operation at 0.6V. A 65nm library characterized at a supply voltage of 0.81V will be used to synthesize the designs at the weak corner, and the corresponding area, power, and timing results will be presented. An equivalent clock period of 23 ns for the synthesis conditions will be used as the target in discussions regarding critical path delay.

Parallelism is only possible when the units of video data being processed are independent. For entropy coding in VC-1 and H.264, macroblock parallelism is not possible because there are interdependencies between macroblocks in the same frame. For CABAC in H.264

there is two additional sources of serial dependency, context-adaptivity and interval subdivision, which are discussed in more detail in Chapter 4. Based on these constraints, the smallest stand-alone unit of data that can be processed by entropy coding is a single frame. Thus, three-frame parallelism is still possible but macroblock parallelism is not. Additional techniques are required when designing the interface between the entropy coding block and all other functional blocks supplying the data to be compressed in order to synchronize and reorder the data into the proper format for entropy encoding, but these design challenges are beyond the scope of this thesis and will not be discussed.

1.6 Engineering Trade-off Differences for VC-1 and H.264

The H.264 entropy encoding algorithm has a higher complexity than that of VC-1 in terms of its highly serial behaviour due to bin-to-bin dependencies. The recursive interval subdivision steps for H.264 cannot be parallelized in the way that table lookups for VC-1 can. As such, it is reasonable to expect the operations in H.264 entropy encoding will require a much longer delay than VC-1. With system level integration of the collaborative work on the video encoder implementation in mind, the operating frequency is held constant across the two standards. Since the overall operating frequency of the dual-standard entropy encoder is bound by the longest critical path delay, and the H.264 block must necessarily contain that longest critical path, we identify that timing is the most important metric for H.264 entropy encoding in this dual-standard block. If the critical path in H.264 exceeds the overall clock period, the system throughput requirements cannot be fulfilled. Thus area and power penalty are acceptable as long as critical timing can be met. VC-1 entropy encoding, on the other hand, does not contain the dominant timing path, so more freedom is available for area and power optimizations at the expense of timing penalty.

1.7 Thesis Contribution and Organization

This thesis examines the challenges in the design and implementation of a dual-standard entropy coder for video processing applications. Through researching previous work on various sets of design specifications, and examining the new standard profile and throughput constraints, the contributions of this thesis are discussed and organized in the following manner.

Chapter 2: Entropy Encoding in VC-1

- A general survey of the VC-1 video compression standard as it pertains to entropy encoding, and the derivation of specifications for this design.
- A discussion of the reference software simulations performed to determine the typical data pattern that can be exploited in the hardware design.

Chapter 3: Entropy Encoder Hardware Architecture for VC-1

- Use of parallelism at the syntax element level and resource sharing to minimize the area and power overhead from parallel architecture while achieving throughput requirements.
- Synthesis results reporting area and power breakdown and critical timing path of the implemented hardware and quantifying the area and power savings from the optimization techniques employed.

Chapter 4: Entropy Encoding in H.264

- A discussion of entropy coding in the H.264 video compression standard.
- A description of the approximations and reference software simulation results used to define the requirements of the module design.

Chapter 5: Entropy Encoder Hardware Architecture for H.264

- Design decisions for a pipeline architecture that eliminates data dependencies across stages to mitigate throughput degradation from stalling.
- Architectural optimizations to reduce the timing cost of multi-symbol encoding.
- Synthesis results reporting area breakdown and area overhead from optimizations to reduce the large critical timing path.

Chapter 6 draws several conclusions to this work and discusses further optimizations and features that may be explored in future work.

Chapter 2

Entropy Encoding in VC-1

The SMPTE 421M video codec standard documented in [3] is a standard video format released in 2006. Informally known VC-1, it originated from a proprietary video format by Microsoft. This standard was selected as one of the two standards supported in this low-power video encoder design because it is emerging as a popular alternative to the H.264 standard. It is supported by a wide range of platforms including HD DVD, Blu-ray Disc, Xbox 360, and Playstation 3. The level of feature and specification dissimilarity between the two standards poses an interesting research problem when designing a reconfigurable system, in particular one that minimizes power consumption and possibly area.

The VC-1 standard includes three profiles and up to five levels in each profile to support a wide range of resolution and frame rate requirements, from 176x144 @ 15Hz (QCIF) to 1920x1080 @ 60Hz (1080p). A profile defines a subset of coding tools and compression algorithms featured in a particular encoder or decoder, while multiple levels are defined to place a set of constraints on the various parameters available within each profile. As it pertains to entropy coding, the profile defines a subset of the bitstream syntax specified in the standard, and the level places constraints on the values of those syntax elements present in the profile. There are three profiles (Simple, Main, and Advanced) and varying numbers of levels for each profile defined in the VC-1 standard [3]. Since the target resolution and frame rate of the video encoder is 4096x2160 (4Kx2K) @ 30Hz, which corresponds to two times (four times the number of pixels at half the frame rate) the maximum bit rate supported by VC-1, the targeted profile and level is Advanced Profile Level 4.0, the highest profile and level defined in the standard (i.e., contains all compression features defined).

The entropy coding scheme employed by the VC-1 standard is static variable length coding. Along with the bitstream format, one or more codeword tables for each type of data, known as a syntax element, are defined in the standards document [3]. The definition of more than one table provides the ability to adapt the probability model for certain syntax elements commonly based on the quantization (QP) value used for encoding and in one case based on BFRACTION. Tables for other syntax elements are selected by signaling an additional syntax element and allows the encoder implementation to select the appropriate table. The syntax elements that are encoded using variable length codeword (VLC) tables are listed along with the number of tables designated for each and the maximum number of entries in each group of tables in Table 2.1. Some examples of syntax elements with multiple VLC tables include motion vectors and transform DC coefficients. In cases where more than 1 table is defined, the method of table selection is also listed.

Table 2.1: VC-1 syntax elements and their table selection method.

Syntax Element	**Number of Tables**	**Table Size**	**Selection Method**
PTYPE	1	5	-
MVRANGE	1	4	-
MVMODE	2	5	Based on QP
BFRACTION	1	21	-
I_CBPCY	1	64	-
P_CBPCY	4	64	Selected by encoder
TTMB	3	4	Based on QP
BMVTYPE	2	3	Based on BFRACTION
ESCMODE	1	3	-
ESCLVLSZ	3	11	Based on QP
TTBLK	3	8	Based on QP
SUBBLKPAT	3	15	Based on QP
IMODE	1	7	-
QUANTMODE	1	12	-
MVDATA	4	73	Selected by encoder
DCCOEF	4	120	Selected by encoder
ACCOEF	4	186	Selected by encoder and based on QP

2.1 VC-1 Video Standards Study

The standards document [3] specifies the bitstream format organized in the following hierarchical grouping: sequence layer, entry-point layer, picture layer, slice layer, macroblock layer, and block layer. The Advanced Profile has a bitstream format that includes all of these layers (as opposed to the Simple and Main Profiles which omit the entry-point and slice layers). A sequence contains one or more entry-point segments; an entry-point segment contains one or more pictures; a picture consists of one or more slices; and a slice consists of many macroblocks, each of which is made up of four luma (monochrome brightness) blocks with dimensions of 16 pixels by 16 pixels and two chroma (colour-difference) blocks. This hierarchical structure is illustrated in Figure 2-1. In the picture layer, all slices except for the first slice in the picture are encoded with a slice layer to indicate slice-specific information such as row address of the first macroblock row in the slice.

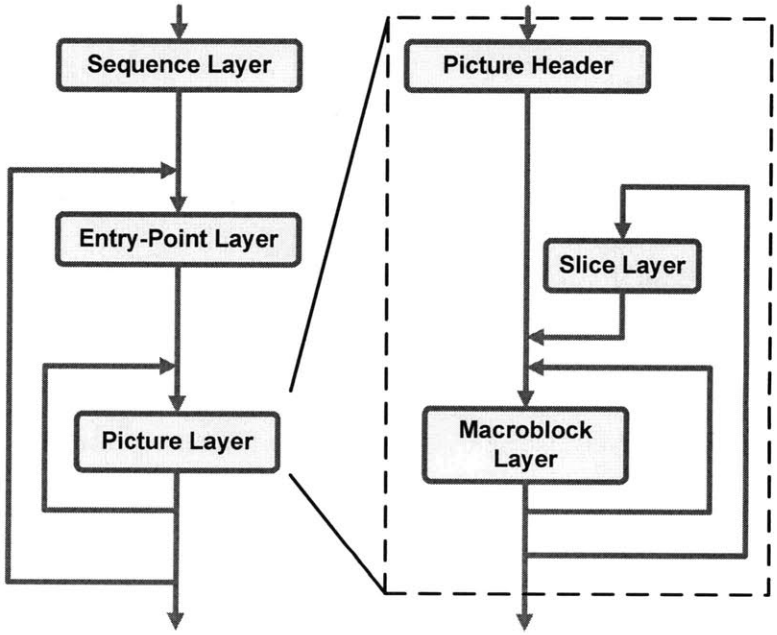

Figure 2-1: Bitstream structure for VC-1 Advanced Profile.

There are some interdependencies between syntax elements both within the same layer and across layers. Specifically, the values of certain syntax elements dictate whether a subsequent syntax element appears in the same layer, while others are defined as the set indices for subsequent syntax elements that have more than one VLC table. For example, if the syntax element *DCCOEF* (representing the transform DC coefficient) in the block layer

is zero, the syntax element *DCSIGN* is not present in the block layer. Likewise, the syntax element *TRANSDCTAB*, which appears in the picture header, determines which of the two defined table is used to encode (and decode) *DCCOEF* in the block layer. It is noted that set indices need to be stored as they are encountered in the bitstream and subsequently retrieved when encoding later syntax elements that depend on their values.

There is also a distinction to be made between syntax elements that are represented by variable length coding and others that appear in the bitstream with a fixed length. Typically these fixed length codes appear immediately after a variable length code in the bitstream. Some examples of these *VLC/FLC* pairs include *DCCOEF/DCSIGN* and *ACCOEF1/LVLSIGN*. Both of these pairs have a large frequency of occurrence based on statistics generated from typical video sequences, and the process of generating this data is described in Section 2.3.

2.2 Specifications

The entropy encoder design is targeted for fabrication as part of collaborative work with other group members on a low power dual-standard video encoder for ultra high definition (ultra-HD) video content. Power consumption is reduced by operating at voltages as low as 0.6V, but this also reduces the operating frequency and places a constraint on the performance of the encoder. As such, the VC-1 entropy encoder must meet a certain throughput level given this performance constraint through parallelism. By processing multiple syntax elements at the same time, the number of cycles required to encode the video content is reduced, and real-time encoding at the target frame rate of 30 frames per second can be achieved.

To illustrate the procedure for determining the throughput requirement for the VC-1 entropy coder, the following formulation is used. For a given video sequence, the content is processed by other functional blocks in the video encoder such as transforms and motion compensation and passed to the entropy encoder in units called syntax elements, as discussed in Section 1.2. The entropy encoder in turn must process these syntax elements and generate a bitstream representation of the data to be transmitted or stored. Thus, as the *last* block of the encoder, the entropy encoder is responsible for supplying the output of the encoder, which must meet the maximum bitrate (i.e., number of bits per second)

requirement as specified in the video standards document. For Advanced Profile Level 4.0, the maximum bit rate requirement is 135 megabits per second (Mbps).

The entropy encoder is similarly constrained by the syntax elements at the input. Since the overall frame rate of 30 frames per second must be maintained, the peak number of syntax elements to be processed per frame dictates how many syntax elements need to be processed per cycle in the worst case. Assuming all frames in a sequence are encoded serially at an operating frequency of 25MHz, the requirement on number of syntax elements encoded per cycle can be estimated by a simple calculation using reference software simulation numbers, as discussed in Section 2.2.1. This computation is described in Equation 2.1.

$$\begin{aligned} \text{Number of SEs per Cycle} &= \frac{\text{Number of SEs per second}}{25 \text{ MHz}} \\ &= \frac{\text{Number of SEs per frame (peak)} \times 30 \text{ frames/second}}{25 \times 10^6 \text{ cycles/second}} \end{aligned} \tag{2.1}$$

2.2.1 Number of Syntax Elements to Process

To compute the minimum number of syntax elements that need to be processed per cycle, an *se_count* variable is added to the VC-1 reference software encoder and incremented whenever bits are generated from encoding syntax elements. At the end of running the encoder software on various typical full-HD video sequences, the count valuable holds the total number of syntax elements processed in each frame of video content and is reset at the start of each frame. The number of bins processed per frame is significant because the specification requires a fixed number of frames per second, which translates to a fixed time for each frame to be encoded. The peak values for the 1080p standard sequences horsecab, waterskiing, rally, and splash are shown in second column of Table 2.2. The distribution of the bin count per frame is plotted in Figures 2-2 to 2-5.

Assuming a target frequency of 25MHz, and that the number of syntax elements approximately scales linearly with the number of pixels in a frame, the average rate of syntax element processing is multiplied by 4 to account for the four-fold increase in number of pixels from the 1920x1080 resolution of the example sequences to the targeted resolution of 4Kx2K. This number is then divided by the target operating frequency of 25MHz to obtain the targeted number of syntax elements to be processed in each cycle, which is tabulated in the third column of Table 2.2. This indirect calculation was necessary since the VC-1

reference software does not accept sequences with resolutions higher than 2Kx2K.

Table 2.2: Common full-HD video sequences, corresponding numbers of syntax elements processed over ten seconds, and numbers of syntax elements processed per cycle assuming target frequency of 25MHz mapped to quad full-HD resolution.

Full-HD Video Sequence	**Number of SEs Per Frame (Peak)**	**Number of SEs to be Processed Per Cycle**
Horsecab	4630704	5.56
Rally	2739984	3.29
Splash	3436616	4.12
Waterskiing	3197772	3.38

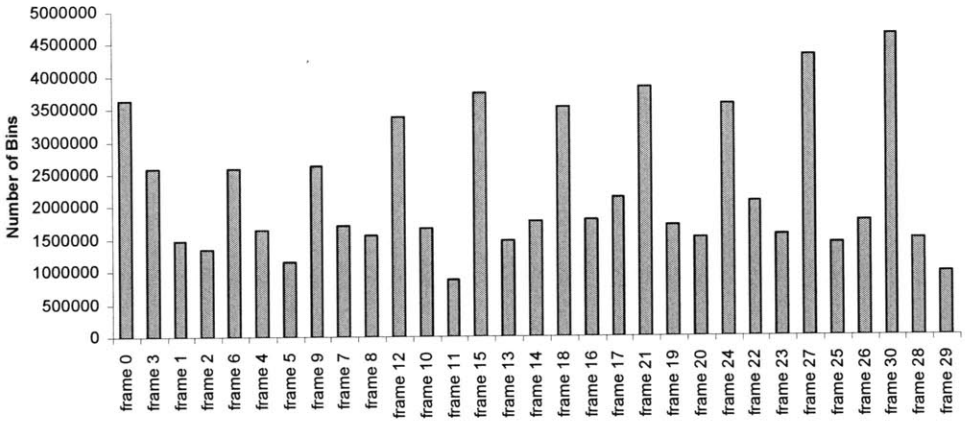

Figure 2-2: Number of bins per frame for the *Horsecab* sequence.

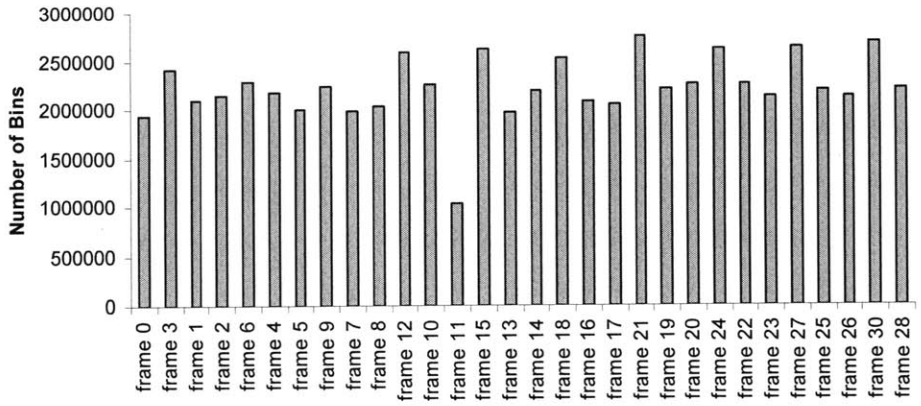

Figure 2-3: Number of bins per frame for the *Rally* sequence.

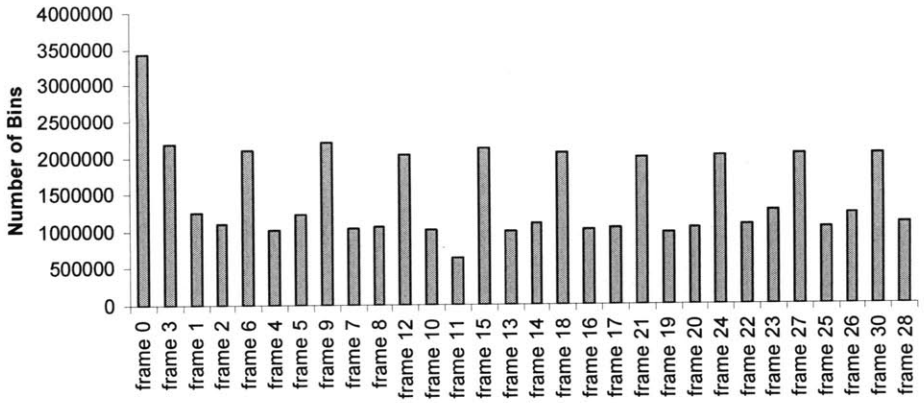

Figure 2-4: Number of bins per frame for the *Splash* sequence.

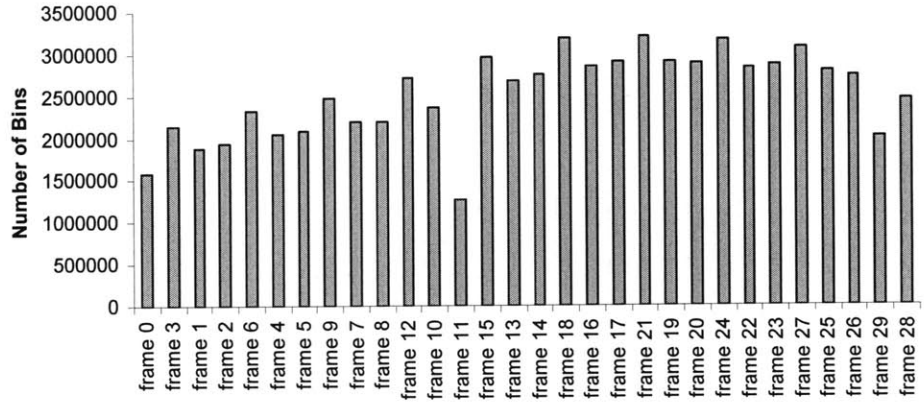

Figure 2-5: Number of bins per frame for the *Waterskiing* sequence.

2.3 Simulation with Reference Software

To determine the required throughput to guarantee the specifications listed in Section 2.2, several modifications are made to the VC-1 reference software. By generating statistics and likely patterns from encoding typical video sequences using the reference design, we can specify a set of targets for the architectural design of the VC-1 entropy encoder block. These targets include the most frequently occurring individual syntax elements and the most likely groupings of syntax elements. The following sections detail the process by which these statistics are generated, and the ways in which they are applied to the VC-1 block architecture described in Chapter 3.

2.3.1 Frequency of Syntax Element Occurrence

To identify the frequently occurring syntax elements, a count variable for each type of syntax element is added to the reference software and incremented every time that type of syntax element is encoded. Figures 2-6 to 2-13 shows the proportional frequencies so that a comparison can be made as to their relative importance. This data is obtained from encoding four standard full-HD video sequences at two different quantization levels, as noted in the captions. Since it can be shown that the most likely syntax elements are ACCOEF1, LVLSIGN, DCCOEF, and DCSIGN, it is reasonable to design the parallel architecture based on these most likely combinations. Considerations such as table usage (for shared/local table categorization described in Section 3.2) also take into account that ACCOEF1 and DCCOEF can occur many times consecutively.

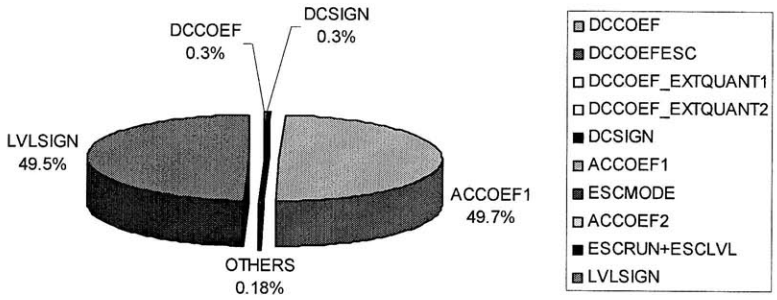

Figure 2-6: Relative frequencies of occurrence for syntax elements in high quality *Horsecab* sequence.

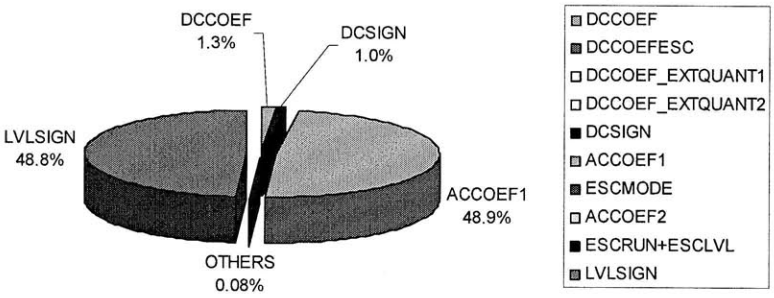

Figure 2-7: Relative frequencies of occurrence for syntax elements in low quality *Horsecab* sequence.

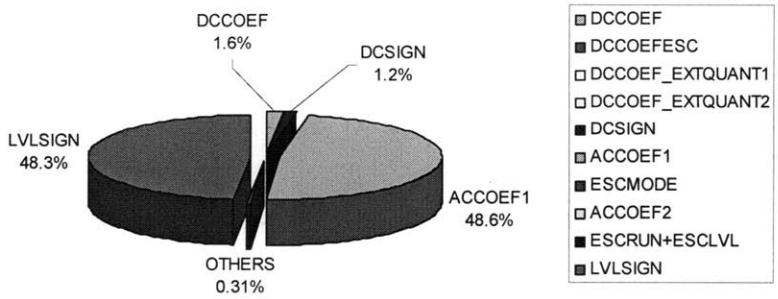

Figure 2-8: Relative frequencies of occurrence for syntax elements in high quality *Rally* sequence.

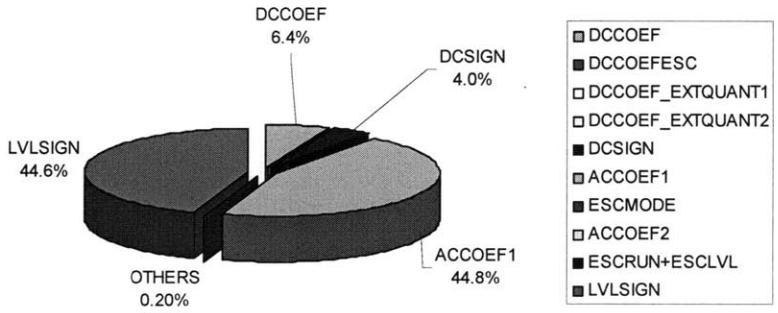

Figure 2-9: Relative frequencies of occurrence for syntax elements in low quality *Rally* sequence.

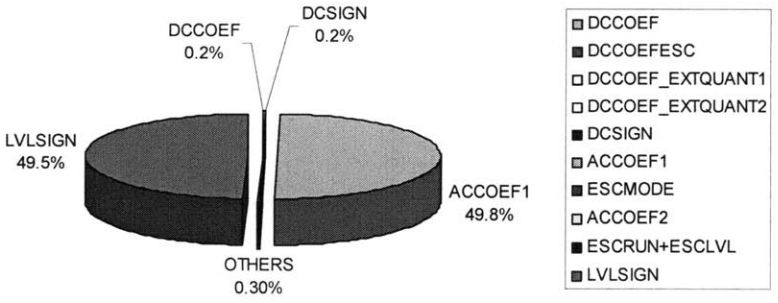

Figure 2-10: Relative frequencies of occurrence for syntax elements in high quality *Splash* sequence.

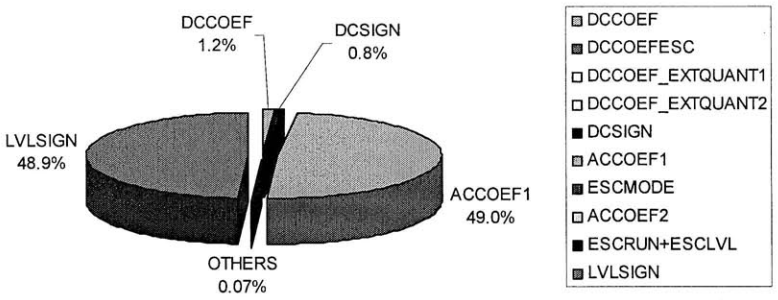

Figure 2-11: Relative frequencies of occurrence for syntax elements in low quality *Splash* sequence.

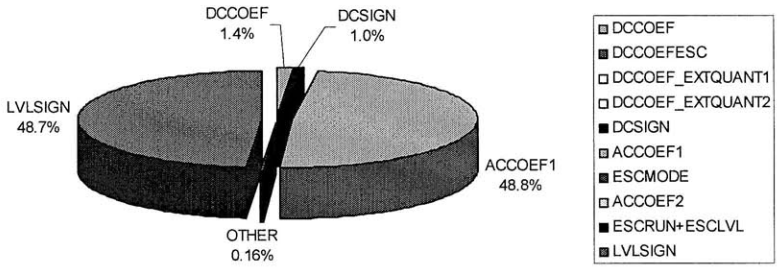

Figure 2-12: Relative frequencies of occurrence for syntax elements in high quality *Water-skiing* sequence.

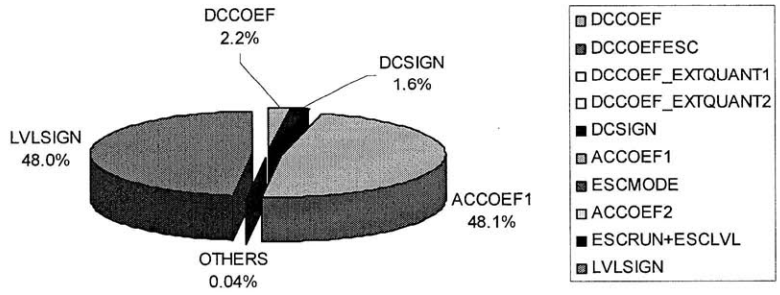

Figure 2-13: Relative frequencies of occurrence for syntax elements in low quality *Water-skiing* sequence.

2.3.2 Patterns of Neighbouring Syntax Elements

Since we would like to encode multiple syntax elements in parallel, any common patterns by which particular syntax elements are encoded can be exploited when designing the en-

coder architecture. For instance, if a particular group of syntax elements always occurs together and occurs frequently, the entropy encoder block can be designed to assume this pattern most of the time and disable unused portions in other less likely cases. Thus it is constructive to investigate patterns near the most frequently occurring syntax elements. From Figures 2-6 to 2-13, the four most frequently occurring syntax elements are ACCOEF1, LVLSIGN, DCCOEF, and DCSIGN. By dumping patterns from the reference software while encoding standard video sequences and examining the standards document [3], it is shown that ACCOEF1 can either be followed by ESCMODE or LVLSIGN, and DCCOEF can either be followed by DCCOEFESC, DCCOEF_EXTQUANT1, DCCOEF_EXTQUANT2, or DCSIGN. From the relative frequency of occurrence between these candidates for the next syntax element following ACCOEF1, LVLSIGN has a 98% chance of occurring after ACCOEF1 and dominates the pattern, rendering the *VLC-FLC* pairing between these two syntax elements a highly likely event in the sequence. The same can be seen for DCCOEF, where DCSIGN has a 50% chance of occurrence (due to the fact that DCSIGN is not coded if DCCOEF = 0).

Chapter 3

Entropy Encoder Architecture for VC-1

The specifications and design constraints discussed in Chapter 2 lead to the requirement that at least three syntax elements be processed per clock cycle. This chapter outlines a parallel architecture that can process between three and six syntax elements in the same cycle, depending on the bitstream structure of the section being encoded.

3.1 Architecture Design

The VC-1 entropy encoder generates a serial bitstream representation from syntax elements that describe the video content. These syntax elements are supplied by the other functional blocks in the encoder, and are defined in the standards document. Entropy encoding in the VC-1 standard is based on variable length coding, and additional flexibility is provided by the ability to select one of several pre-defined variable length code (or VLC) tables when encoding each specific syntax element. In terms of general variable length coding, each entry in these tables represents one of the leaf nodes of a Huffman tree generated in the manner discussed in Section 1.3, such that the shortest code words represent the values with the highest probabilities of occurrence, and the longest code words represent those with the lowest probabilities of occurrence.

In a hardware implementation, these pre-defined *VLC tables* are hard-coded as read-only memories (ROMs) on the system. These can logically be organized by syntax elements and addressed by the particular value of the syntax element being encoded. The output of

these ROMs are generated using multiplexers, and contain the codeword corresponding to that syntax element value, as well as its length - since the codes are of variable length, the number of bits in a particular codeword must be explicitly signaled for bit-packing purposes. The data structure by which this information is represented in each entry of the VLC tables is discussed in Section 3.2.

To encode more than one syntax element at once, the design needs to include parallel encoding cores that process data simultaneously. Each of these parallel cores need to access the VLC tables to determine the appropriate codeword from the given syntax element value, so the static tables should be duplicated such that one copy is assigned exclusively to each encoding core. Section 3.2 further describes the optimizations made to reduce the redundancy of duplicating tables.

Of all the syntax elements defined in the VC-1 Standards document, two main categories can be formed: variable length codes and fixed length codes. As explained above, variable length codes (VLCs) are all syntax elements that require a table lookup in the encoding process, where the value of the syntax element, given in some fixed length representation, is mapped into a code of variable length. Fixed length codes (FLCs), on the other hand, are syntax elements that are always represented by a fixed number of bits in the output bitstream, and thus can be encoded directly without any table lookup. Typically the syntax elements that have a large range of values or occur the most frequently in the bitstream syntax are coded using VLCs, whereas FLCs are used for syntax elements that are only one or two bits wide, or only occur once in the beginning of the video sequence (i.e., in the sequence or entry-point layers). This distinction between the two types of syntax elements lends itself to the design of a separate module to handle each type - these encoding cores are referred to as the variable length coding (VLC) engine and the fixed length coding (FLC) engine, respectively.

The paired VLC and FLC engines take the type (*syntax_index*) and value (*syntax_entry*) of each syntax element as their inputs and generates the output code (*code_out)* and its corresponding length (*len_out*) as outputs. The concatenation and bit-packing module then takes the outputs from the VLC and FLC engines and combines them into a contiguous bitstream at the top-level output. These modules are described in more detail in the sections to follow.

The simulation (outlined in Section 2.3.2) performed using the reference software in-

dicates that neighbouring syntax elements are likely to be opposite types - i.e. syntax elements often occur in *VLC-FLC* pairs. By identifying these pairs and scheduling them to be encoded in parallel, the number of syntax elements encoded per cycle is between one (in the case that a *VLC* syntax element is followed by another *VLC* syntax element, or same with *FLC*s) and two (in the ideal case that the *VLC* syntax element is immediately followed by an *FLC* syntax element). To guarantee at least three syntax elements are encoded per cycle, six encoding engines (designating three for *VLC* syntax elements and three for *FLC* syntax elements) are implemented as parallel blocks, as illustrated in Figure 3-1. Depending on the pattern of the incoming syntax elements, up to three (in the case where all three syntax consecutive syntax elements are of the same type) of these parallel engines are disabled to match that pattern. All possible usage scenarios of the *VLC* and *FLC* engines are tabulated in Table 3.1.

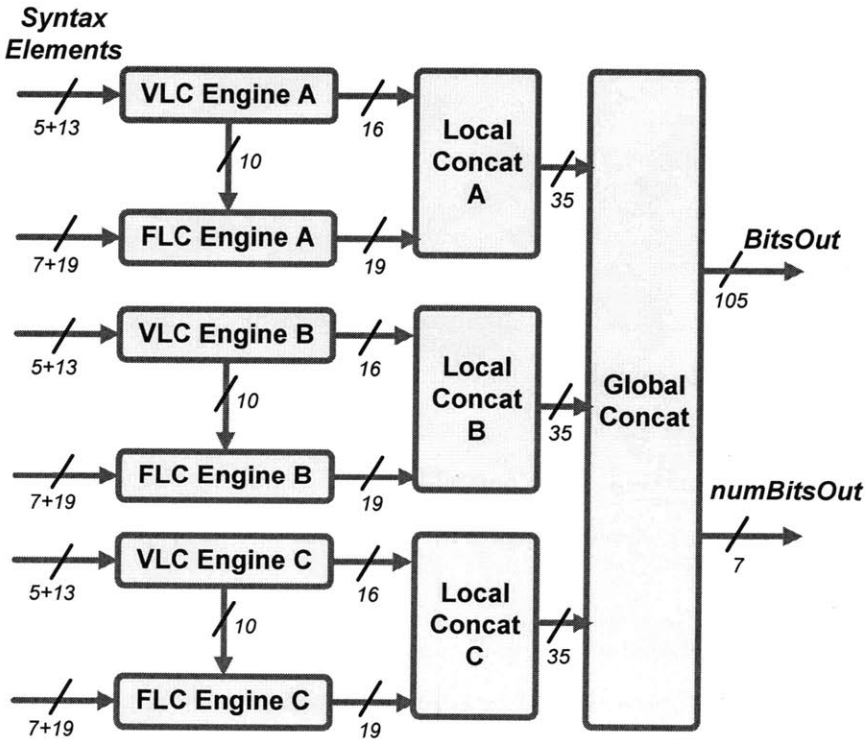

Figure 3-1: Parallel Structure of Three *VLC-FLC* Pairs.

Table 3.1: Patterns and corresponding encoding core usage models.

Pattern	VLC_A	FLC_A	VLC_B	FLC_B	VLC_C	FLC_C	***Num SEs Encoded***
V,F,V,F,V,F,...	1	1	1	1	1	1	6
V,F,V,F,V,V,...	1	1	1	1	1	0	5
V,F,V,F,F,...	1	1	1	1	0	1	5
V,F,V,V,...	1	1	1	0	1	0	4
V,F,F,V,F,...	1	1	0	1	1	1	5
V,F,F,F,...	1	1	0	1	0	1	4
F,V,F,V,F,...	0	1	1	1	1	1	5
F,V,F,V,V,...	0	1	1	1	1	0	4
F,V,F,F,...	0	1	1	1	0	1	4
F,V,V,F,V,...	0	1	1	0	1	1	4
F,V,V,V,...	0	1	1	0	1	0	4
F,V,V,F,V,...	0	1	1	0	0	1	5
F,F,V,F,...	0	1	0	1	1	1	4
F,F,V,V,...	0	1	0	1	1	0	3
F,F,F,...	0	1	0	1	0	1	3

3.1.1 Variable Length Coding Engine

The variable length coding engine generates the bit representation of syntax elements categorized as *VLC*s. Given the type of syntax element to be encoded, the variable length coding engine selects one of multiple predefined codeword tables on which a lookup operation is performed. Among all tables designated to a particular syntax element, the selection is made based on supplemental information such as picture type and B picture fraction in some cases and explicit table selection variables defined earlier in the bitstream. The appropriate table is signaled by a one-hot table selection bus, and an address corresponding to the value of the syntax element is provided to select the appropriate entry within that table.

Since the variable length codes for different entries can have arbitrary length by definition, the result of the table lookup must indicate both the number of bits used in the codeword as well as the actual bit representation. The width of this table output is bound by the longest codeword in the table. A one-hot scheme is used to represent the length of the codeword, thus the bitwidth of each entry in a given *VLC* lookup table is at least equal to and at most twice as long as the longest codeword in that table. For example, in a table where the longest codeword is 10 bits wide, a codeword with value 19 and length 6

should have the binary form of 6'b010011, so its corresponding entry in the *VLC* table is implemented as 20'b00000100000100110000 – the 10 LSBs indicate the codeword aligned to the left, and the 10 MSBs indicate that the codeword ends on the sixth bit (counted from the left). The one-hot scheme was chosen with consideration to the barrel shifter implementation – as discussed in Section 3.1.3, the individual codewords are concatenated into a single block of bits using large barrel shifters. A comparison between two implementations of a barrel shifter is presented in [8]. The regular structure, whose select bits signaling how many bit positions to shift are implemented using the one-hot scheme (n bits for an n-bit-wide barrel shifter), was demonstrated to have a single transistor delay independent of the width of the barrel shifter. The logarithmic structure, on the other hand, takes encoded select bits ($log_2 n$ bits for an n-bit-wide barrel shifter), but has a delay of $log_2 n$ transistors. The performance of the regular structure is thus shown to be better than that of the logarithmic structure.

Of all the tables provided in the standard, the table containing the maximum-sized codeword of 26 bits is the *High-Motion Luma DC Differential* table; thus, the data bus providing the table output to the *VLC* engine is designed as 26 bits wide, with extra bits unused (i.e. the values appearing on these bits are *don't care*) for all other tables that have a smaller maximum codeword size. The probability of extra switching activity introduced by these don't care bits is minimized by setting the unused bits to 0. By multiplexing all outputs of the lookup tables onto a single 26-bit bus, a single input port provides codeword information to the *VLC* engine based on the syntax element type and value queried. This structure is schematically shown in Figure 3-2.

3.1.2 Fixed Length Coding Engine

The fixed length coding engine provides the bit representation of syntax elements categorized as *FLC*s, i.e. all syntax elements whose value can be represented with a fixed number of bits in the VC-1 bitstream syntax defined in [3]. The output of this block is designed to accommodate the longest fixed length code defined in the bitstream structure, which is 19 bits wide. In the straightforward implementation where the variable length coding and fixed length coding modules operate completely independently, this operation involves directly passing the syntax element value and its length to the output, where the value is strictly taken as a codeword and incorporated in the bitstream. In this design, however, for

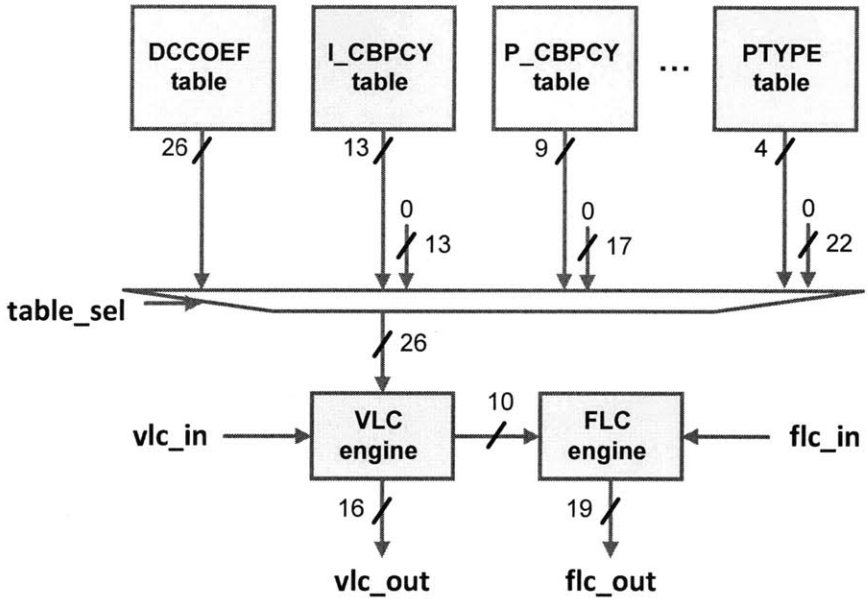

Figure 3-2: Block diagram of connection between the *VLC* tables and the *VLC* engine.

reasons detailed in Section 3.3.1, resources are shared between the *VLC* and *FLC* engine pairs in such a way that there needs to be additional processing to generate the output. Specifically, the *FLC* engine combines 10 bits from the *VLC* engine output with its own bits to form an intermediate, concatenated codeword. This concatenation operation adds to the complexity of the *FLC* engine, but since the functionality of the pure *FLC* engine only involves generating the (fixed) length of the codeword based on the syntax element being processed, this amount of added complexity is reasonable. Moreover, for some syntax elements, while the length of the codeword is fixed, the bit representation does not map directly from the input value. The fixed length coding engine also contains extra logic to handle these cases.

3.1.3 Concatenation and Bit-Packing

After the variable length code and fixed length code mapping performed by the *VLC* and *FLC* engines, the individual bit representations of multiple syntax elements need to be combined into a single contiguous bitstream at the output. In this particular implementation the number of codewords to be combined equals the number of syntax elements processed in a given cycle, which can be a number between three and six, inclusive. This procedure is performed in two steps.

First, in the concatenation step, the codewords are shifted to the appropriate positions and a bit-wise *OR* operation is performed to combine them into a single block of bits to be sent to the output. As with the individual codewords, the length of this block must be specified as it can vary between three (in the case of three *FLC*s described in the last row of Table 3.1, each one bit long) and 81 bits (in the case of three *VLC-FLC* pairs of the longest DCCOEF codeword and a DCSIGN flag – there are no other *VLC-FLC* pairs that exceed this width). In the straightforward implementation, this is achieved by zero-padding each codeword to the maximum combined number of bits and performing a shift operation to each codeword before feeding them into a six-input *OR* gate. In this case, this number is three times the sum of the output widths of single *VLC* and *FLC* engines, i.e. $3 \times (26 + 19) = 135$ bits. A barrel shifter is used to perform the shift by an arbitrary number of bits in a single cycle. The width of this shifter must again match the 135 bits. Since the overall architecture assumes a fixed ordering between the outputs of the various engines (i.e. VLC_A is followed by FLC_A then VLC_B and so on, where unused engines are simply disabled in cases where the sequence of syntax elements at the input does not match this pattern), the concatenation can be performed hierarchically to reduce the width of each barrel shifter and thus area consumption, as described in detail in Section 3.3.2.

Given this combined block of bits and its length, the bit-packing step is performed, where the bits are written to a FIFO buffer. This step is necessary to average the throughput of the bitstream output, as the number of bits written in any given cycle varies, but the output should read a constant number of bits out every cycle, at the output of this FIFO. The proper operation of the bit-packing step also relies on the ability to stall the engine from sending more bits to the output FIFO once it is full. This is handled using a single-bit FIFO full signal fed back to the entropy encoder block.

3.2 Table Optimization Techniques

The lookup tables containing the mapping from syntax element values to variable length codewords provide the information needed by each *VLC* engine to generate an output. Thus, the entire set of tables is replicated for each parallel *VLC* engine. However, since only one table is accessed at a time, some techniques can be applied to reduce the amount of redundant data in the tables by exploiting data patterns and statistics. By categorizing

tables into two groups, where one group is shared among all three *VLC* engines and the other is duplicated for each, the total table size can be reduced. Additional area reduction can be achieved by further partitioning some of the local tables based on probability of occurrence of the individual syntax element value corresponding to a codeword. Interconnect cost is minimal since only three *VLC* engines share access to a set of shared tables – this means that the added area cost on both the addressing and output ends of the shared tables is at most that of a three-to-one multiplexer.

3.2.1 Table Sharing

Table sharing refers to the identification of low-usage lookup tables for variable length codes and reusing a single copy of the table between three *VLC* engines. Since the six-way parallel architecture allows a maximum of three *VLC*s to be encoded in the same cycle, the extreme approach of sharing all codeword tables would cause problems when more than one of the three parallel table lookups query the same table for different table entries. Assuming the hardware is implemented such that all three tables can still receive the output of any table lookup, the fact that the multiple queries to the same table can be for different addresses can result in one of two undesirable outcomes depending on the implementation of the address bus. In this implementation, potential contention on the address bus is avoided by multiplexing the addresses provided by the three *VLC* engines in a prioritized fashion, such that one *VLC* overrides the other two when multiple engines try to address different entries in the same table (this structure is schematically shown in Figure 3-3. However, the *VLC* engine(s) given lower priority would read a codeword that does not match the one it tries to address. This type of data collision (where multiple tables fight for use of a lookup table) must be mitigated in the final design. As each table is directly tied to a single syntax element, it is sufficient to check that the same syntax element is not issued more than once to the three parallel *VLC* engines.

As illustrated in Figure 2-1, the bitstream representation of a video sequence is organized in hierarchical layers, with multiple instances of a lower layer being encoded between each higher layer to delineate all the data contained in that higher unit. The picture layer representing a single frame in the video sequence, for instance, is comprised of a single picture header followed by many instances of the macroblock layer. Each macroblock maps to a 16x16 pixel block of luminance data, so each 4Kx2K frame contains 32768 macroblocks

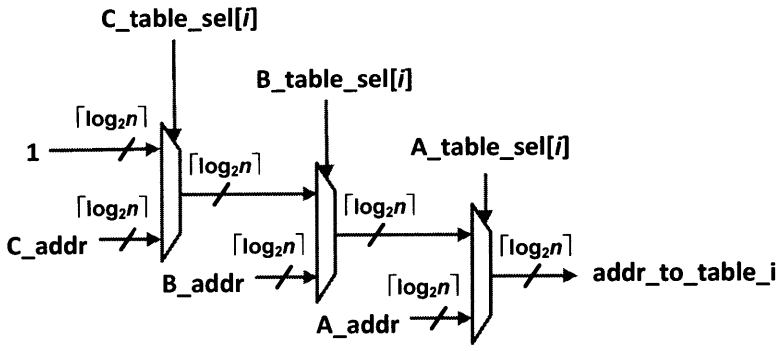

Figure 3-3: Address handling for shared tables (table indexed by i, with n entries).

by the calculation illustrated in Equation 3.1. This large number of macroblock layers to be encoded between consecutive picture headers acts as a buffer that guarantees the same syntax element in the picture layer cannot be found in any window of length three. This essentially means that all *VLC* lookup tables pertaining to picture layer syntax elements can be categorized as shared without introducing data collisions in any given cycle.

$$\begin{aligned} \text{Number of MBs per Frame} &= \frac{4096 \text{ pixels}}{16 \text{ pixels}} \times \frac{2048 \text{ pixels}}{16 \text{ pixels}} \\ &= (256) \times (128) \text{ macroblocks} \\ &= 32768 \text{ macroblocks} \end{aligned} \tag{3.1}$$

By further examining the bitstream structure in the VC-1 standards document, it can be seen that several other syntax elements only occur once in the layer structure to which they belong and are thus also guaranteed not to reoccur within the three-syntax-element window imposed by the six-way parallel architecture. Designating these tables to be shared results in an area reduction proportional to the ratio between shared and local tables. Specifically, the area saved is equal to approximately twice the collective size of all shared tables. Table 3.2 lists all *VLC* syntax elements along with the syntax layer to which they belong and whether they are categorized shared or local. The area savings demonstrated in synthesis is lumped with that of the table partitioning optimization and summarized in Section 3.2.3.

Table 3.2: Shared versus local categorization of *VLC* syntax elements.

Syntax Layer	Syntax Element	Shared / Local
Picture	*PTYPE*	Shared
	MVRANGE	Shared
	BFRACTION	Shared
	MVMODE	Shared
	MVMODE2	Shared
Macroblock	*I_CBPCY*	Shared
	P_CBPCY	Shared
	MVDATA	Local
	TTMB	Shared
	BLKMVDATA	Local
	BMV1	Local
	BMVTYPE	Shared
	BMV2	Local
Block	*DCCOEF*	Local
	ACCOEF1	Local
	ESCMODE	Shared
	ACCOEF2	Local
	ESCLVLSZ	Shared
	TTBLK	Shared
	SUBBLKPAT	Shared
Bitplane	*IMODE*	Shared

3.2.2 Table Partitioning

For syntax elements that do not satisfy the above requirements, it is not possible to categorize the corresponding *VLC* lookup tables as shared without introducing data hazards. For example, the syntax elements DCCOEF and ACCOEF1 are very likely to occur many times consecutively, such that if the corresponding tables were shared in their entirety, the three parallel *VLC* engines are highly likely to access the shared tables in the same cycle. However, in the case of very large tables where there is a large range in codeword lengths, it may be possible to partition a subset of codewords from those tables into the shared category. This optimization is reasonable because the codeword lengths are inversely related to the probability of occurrence of the corresponding value by definition of variable length coding – the longer the codeword, the less likely its corresponding value will occur in the bitstream, and vice versa. This assumption must be valid for compression to happen. It can be shown that the event of a data collision, i.e., of at least two *VLC* engines to query a long

codeword of the same table, is highly unlikely. This is illustrated by Equation 3.2.2. Let us denote A as the event of a *VLC* engine queries an entry in the table to be partitioned, and B as the event of that entry being categorized as long (i.e. with a codeword whose length exceeds some pre-determined threshold). The joint probability between events A and B equals the probability that a single *VLC* engine selects a long codeword from that designated table. When at least two *VLC* engines select a long codeword from the same designated table, a data collision occurs. Thus the probability of a data collision given that the long codewords are partitioned to a shared table is as defined in Equation 3.2.2.

Event A: One *VLC* engine queries an entry in a specific table

Event B: The entry being queried in that table is *long*

$$
\begin{aligned}
P(B \mid A) &= \frac{P(B \bigcap A)}{P(A)} \\
P(A \bigcap B) &= P(B \mid A) \times P(A) \\
P(\text{data collision}) &= 3 \times P(A \bigcap B)^2 + P(A \bigcap B)^3
\end{aligned} \tag{3.2}
$$

For example, assuming $P(A) \cong 20\%$ and $P(B \mid A) \cong 10\%$, we obtain that $P(A \bigcap B) \cong 2\%$. This gives a probability of 0.12% for a data collision to occur. While this number cannot be taken directly to mean that no collision will occur since the values for $P(A) and P(B \mid A)$ are strictly estimates, one can perform simulations to detect the occurrence of data collision. By carefully selecting the threshold codeword lengths at which to partition the long tables and checking in simulation that no collision occurs, we are able to further reduce the size of the local tables of which each *VLC* engine owns a copy.

An additional bonus saving that comes with table partitioning is the reduction in the maximum codeword length in the local tables. The benefit of this reduction can be better explained with the overall hardware implementation of the individual *VLC* engine in mind. The *vlc_engine* module contains logic that maps any syntax element in the *VLC* category into its corresponding variable length codeword and code length. Given a syntax element type and its value, and in some cases supplementary information such as quantization, the *VLC* engine signals the appropriate table using a one-hot data bus (which each bit enabling one particular table), indicates the syntax element value for which the mapping is to be

done through an address bus and reads the lookup result as an input. The lookup result is provided as a multiplexed output of all tables. In order to accommodate the differentiation between shared and local tables, two sets of these output and input ports are designated for the two table groups. Since they are all multiplexed into an input to the *vlc_engine* module, the width of the lookup result bus for each table is bound by the width of the widest table. By partitioning the long tables where there are large discrepancies between the widths of the shortest and longest codewords, the amount of zero padding needed for the shortest codewords as well as the width of the databus delivering the output of the local tables can be reduced. This bitwidth reduction is illustrated in Figure 3-4.

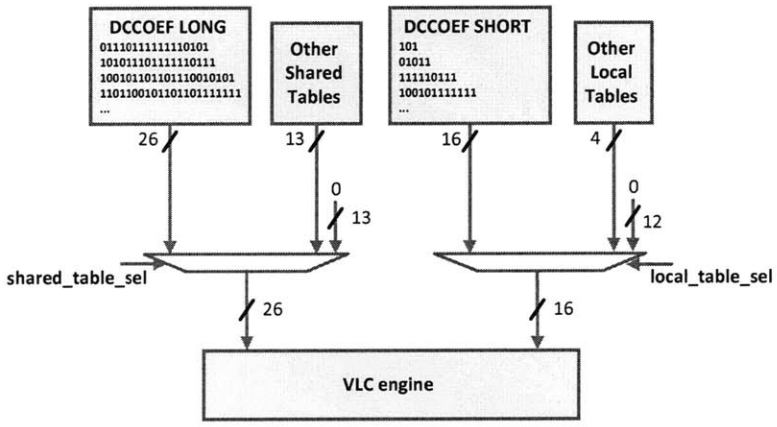

Figure 3-4: Bitwidth reduction for local table output with table partitioning.

3.2.3 Area Savings

By categorizing several whole tables and portions of tables as shared between the three *VLC* engines, area reduction is achieved. From synthesis results, the shared tables have a total gate count of 2.48K whereas the three copies of local tables have a combined gate count of 13.42K. This yields a combined gate count of 15.9K for all *VLC* lookup tables. Comparing to the alternate implementation where all tables are duplicated such that each *VLC* engine uses a designated copy, which would have a total table gate count of $13.42K + 3 \times 2.48K = 20.86K$, the reduction in gate count is $20.86K - 15.9K = 4.96K$ (or 31% of the total table gate count).

3.3 Bit Packing Techniques

Once the individual *VLC* and *FLC* engines have produced a codeword of arbitrary length as an output, these individual groups of bits need to be packed into a contiguous bitstream and sent to the output. As seen in Section 3.1.3, the straightforward implementation requires large amounts of zero padding and very wide barrel shifters. The following two techniques are used in this work to reduce the width of the barrel shifters by performing concatenation hierarchically.

3.3.1 Bitwidth Sharing

From reference software simulations described in Section 2.3.2, we can identify several common *VLC/FLC* pairs. It is noted that while the longest fixed length code is 19 bits wide and fixes the width of the *flc_engine* module output at 19 bits, this longest codeword only occurs when there are no neighbouring *VLC* codes (i.e., when the paired *vlc_engine* is disabled based on the input syntax element pattern. In fact, there are at least ten unused bits (that become zero padded) for *FLC*s that commonly come in after long *VLC*s. For example, the variable length *DCCOEF* syntax element, which corresponds to the tables with the longest codewords of width 26 bits, are always followed by a fixed length *DCSIGN* syntax element, which is a single bit indicating the sign of the preceding DC coefficient. Since we know those extra bits at the output of the *flc_engine* are strictly redundant and will be removed in the concatenation stage, we can reuse them to deliver ten of the 26 bits from the output of the *vlc_engine* by adding some simple logic between the two parallel engines. As shown in Figure 3-2, the *vlc_engine* passes the 10 LSBs from the multiplexed table output to the *flc_engine*. This reuse requires an intermediate concatenation operation between up to 10 bits from the *vlc_engine* and up to 9 bits from the *flc_engine*, schematically illustrated in Figure 3-5 and handled within the *flc_engine*. This reduces the total bitwidth of a *VLC/FLC* pair by 10 bits, down to (26-10) + 19 = 35 bits. By eliminating 10 redundant bits per *VLC/FLC* pair, we reduce the width of the top level barrel shifter by 30 bits, since the final concatenation of all six codewords uses at least two barrel shifters with width equal to the sum of the maximum number of bits at the output of each engine. The top-level barrel shifter in this case has a width of 105 bits, as compared to the original 135 bits.

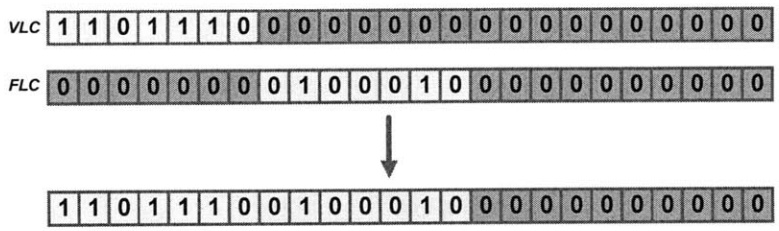

Figure 3-5: Schematic representation of the bitstream concatenation process.

3.3.2 Hierarchical Concatenation

As discussed in Section 3.2, the straightforward implementation to generate the continuous bitstream output involves zero padding all six codewords and shifting each of the codewords (with the possible exception of the "first" codeword which could be read in place) by a variable number of bits before an OR operation is performed to produce the concatenated bitstream where each codeword is immediately followed by the next codeword in the next unused bit position. This operation is schematically shown in Figure 3-5. The six 105-bit-wide values are then fed into a 6-input OR gate to generate a single 105-bit-wide output. The length (or number of valid bits) in the output is determined by summing the lengths of the six codes and provided at the output to indicate to the output FIFO how many bits are being written to the bitstream in this cycle.

The fixed order of the VLC and FLC engines allow hierarchical concatenation to be performed. Suppose VLC_A is combined with FLC_A to form an intermediate codeword $CODE_A$, and so on, such that we have three intermediate codewords $CODE_A$, $CODE_B$, and $CODE_C$ comprised of the original six codewords. A three-input concatenation stage can then take these three intermediate codewords and combine them into a single block of bits to be sent to the output bitstream. The advantage of this hierarchical decomposition is that the sizes of the barrel shifters and idle bits can be reduced. Since the width of a barrel shifter in this context is set equal to the sum of the widths of its inputs, the first level barrel shifter that handles the shifting of FLC_A to be concatenated with the zero-padded VLC_A can be specified to be $16 + 19 = 35$ bits wide. This shifter is shown in Figure 3-6 (the number of bits in each code, which is provided with the actual bits, is not shown here), and an instance of this structure is needed for each of the three pairs of VLC and FLC engines, generating $CODE_A$, $CODE_B$ and $CODE_C$.

Then, at the second stage, two barrel shifters are needed to perform shifting on $CODE_B$

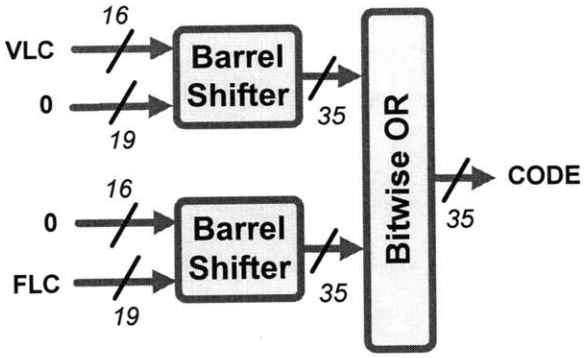

Figure 3-6: Block diagram for *Local Concat* module.

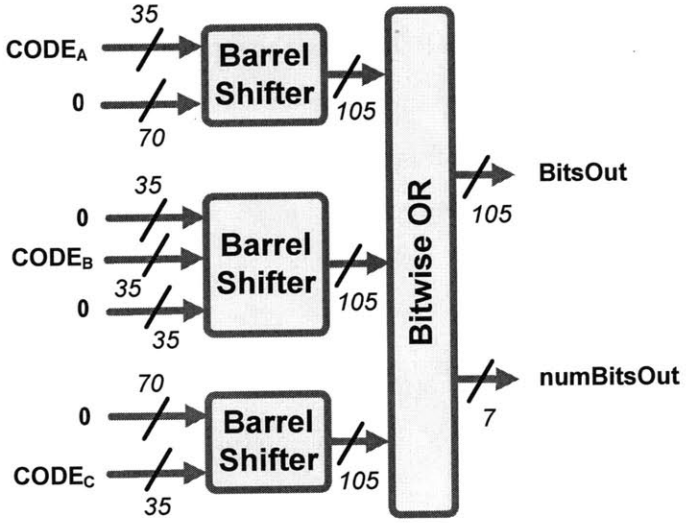

Figure 3-7: Block diagram for *Global Concat* module.

and $CODE_C$, and each is specified at $3 \times 35 = 105$ bits wide, the same size as the barrel shifters in the naive implementation. This structure for combining the three intermediate codewords is illustrated in Figure 3-7 (again the number of bits is provided along with the actual bits in the code, is not shown here for simplicity). However, since both designs require five barrel shifters, but here three of the five is one-third as wide as they were in the first former case, there are savings in terms of area consumption and number of bits on the intermediate databuses, which can reduce overall switching activity of the module.

3.3.3 Area Savings

The top level concatenation consists of two 105-bit-wide barrel shifters and a 3-input, 105-bit-wide bitwise OR block. As shown in synthesis results, the gate count for this module is 43.8K. The three local concatenation blocks are each comprised of a 35-bit wide barrel shifter and a 35-bit wide bitwise OR block. Their combined gate count is 55.21K. Thus the total gate count used for concatenation is 99.01K, which accounts for over 80% of the total gate count of the module. If the straightforward implementation of a single-level concatenation without bit sharing was used, the gate count can be approximately computed as that of five 105-bit-wide barrel shifters and a 6-input, 105-bit-wide OR block. A conservative estimate would be to assume the concatenation module scales with the number of barrel shifters (i.e., assuming the OR block scales the same way with the increased number of inputs). This yields a gate count of $5 \div 2 \times 43.8K = 109.5K$ for the straightforward implementation, which translates to a 10.49K (or 10%) gate count savings using the bit sharing and hierarchical concatenation approaches. It is useful to note that this area reduction might be limited by the module boundaries that exist between the local and top-level concatenation modules.

3.3.4 Power Savings

The power savings from bitwidth sharing and hierarchical concatenation can be approximately computed from synthesis results. The power attributed to concatenation is the sum of power reported for the three local concatenation modules and the global/top concatenation module. This adds to a total concatenation power of 209.7 μW. In the naive implementation, five copies of the equivalent logic as the global concatenation would need to be implemented to concatenate all six codewords by a single OR operation, which maps to a concatenation power of 385.0 μW, so the percentage of power saved is approximately 46%.

3.4 Hardware Verification

To verify the functionality of the hardware, the reference software is used to generate test patterns in the form of inputs and outputs to the entropy encoding module. The hardware should mimic the behaviour of the reference software behaviour, such that the output bitstream can be properly decoded by any standard-compliant decoder.

The test vectors used for verification purposes are designed to map directly to the top-level inputs and outputs of the VC-1 entropy encoder module. As the number of syntax elements processed per cycle can vary from three to six, the test vectors are written to text files one by one and the testbench file reads in as many lines as the number of syntax elements to be processed in the given cycle. The two inputs, *syntax_index* (indicating the type of syntax element being encoded) and *syntax_value* (representing the data to be encoded), and the two outputs, *code_out* (the bitstream representation of *syntax_value*) and *len_out* (the number of bits that this code contains), are written to four separate files, which each line of the file representing the same syntax element. The testbench then reads the same number of lines from each of the four files depending on how many syntax elements are encoded in that cycle. By comparing the output of the hardware module with the software generated output, it was verified across four sequences (10 frames each) that the hardware module does the correct mapping based on the inputs provided. The correctness of the module output also demonstrates that the assumptions about data collisions discussed in Section 3.2 are acceptable, as any data collision would result in the incorrect data being read from the lookup tables and this error would propagate to the output bitstream.

The waveforms displayed in Figure 3-8 show the inputs and outputs of the VC-1 entropy encoder. The six pairs of *index_in* and *entry_in* values correspond to the index and value of the six syntax elements to be encoded in each cycle. The outputs *code_out* (105 bits wide) and *len_out* (7 bits wide) provide the output bitstream representation and the number of bits being sent to the bitstream for each cycle.

3.5 Synthesis Results and Conclusions

The VC-1 entropy encoder module was synthesized using a 65nm library in Synopsys Design Compiler. Since the target operating voltage of the dual-standard video encoder chip is 0.6V, the weak-corner, high-temperature cell library with the lowest characterization voltage available (0.81V) was used for synthesis. The critical path constraint was set to 5 ns (200 MHz), which approximately scales to 20 ns (50 MHz) at a supply voltage of 0.6V. This was intentionally set to be tighter than the targeted 25 MHz to determine the minimum length of the critical path. The total gate count of the module is 136.6K gates, which was obtained by dividing the reported total cell area by the size of the smallest two-input NAND gate as

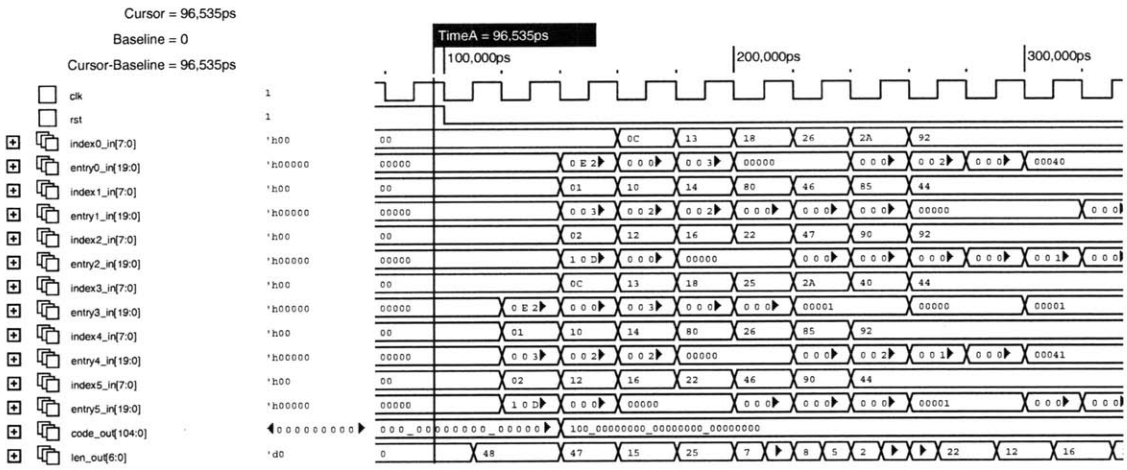

Figure 3-8: Waveform from verification of VC-1 entropy encoder.

defined in the library. The critical path is shown in Figure 3-9, and timing results show its delay to be 6.5 ns.

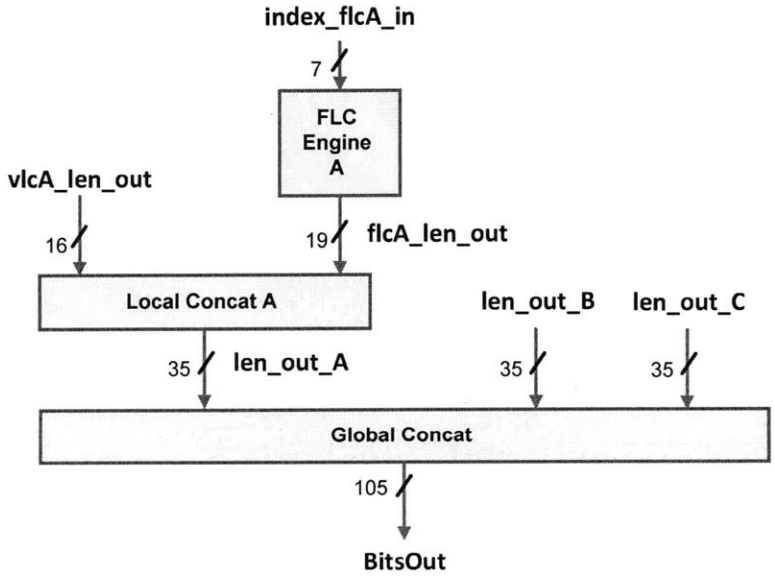

Figure 3-9: Critical path in VC-1 entropy encoder.

On average, the VC-1 entropy encoder processes 4.9 symbols per cycle. Given the maximum delay is 6.5 ns, the clock frequency can be set at 150 MHz. At this frequency, the effective processing rate in terms of syntax elements is found to be 735M syntax elements per second.

3.5.1 Area Distribution

The area of the different modular components in the design are summarized in Table 3.3. The table sharing and partitioning optimizations were shown to have a combined area savings of 4.96K gates (31% of total table gate count), and the optimizations in concatenation was found to yield a combined area savings of 10.49K (10% of total concatenation gate count).

3.5.2 Power Distribution

The hierarchical breakdown of power consumption in the design is tabulated alongside gate count in Table 3.3. The optimizations in concatenation were reported to give a combined power savings of 175.3 μW (46% of total concatenation power consumption).

Table 3.3: Modular area distribution in VC-1 entropy encoder design.

***VLC/FLC* Pair**	**Module**	**Gate Count**	**Power**
A	*VLC* Engine	3.2K	18.3 μW
	FLC Engine	7.1K	5.22 μW
	Local Concat	20.3K	43.1 μW
	Local Tables	7.8K	5.17 μW
B	*VLC* Engine	3.1K	17.1 μW
	FLC Engine	7.1K	5.21 μW
	Local Concat	20.7K	44.8 μW
	Local Tables	7.9K	5.15 μW
C	*VLC* Engine	3.2K	18.3 μW
	FLC Engine	6.5K	5.21 μW
	Local Concat	19.9K	44.8 μW
	Local Tables	7.9K	5.15 μW
Top Level	Top Concat	20.0K	77.0 μW
	Shared Tables	3.7K	10.4 μW
	Total	136.6K	304.5 μW

In summary, parallelism between consecutively encoded syntax elements is used to increase the throughput of the variable length encoder for VC-1. Since the encoding operation is predominantly a static table lookup, parallel processing is possible to any degree up to the point where syntax elements with dependencies (such as those syntax elements that may or may not appear in the bitstream representation depending on the value of a previously encoded syntax element), assuming that area optimization is given a lower priority

than throughput. Here, three syntax elements are encoded in parallel, and the design is optimized for power and area using lookup table sharing and hierarchical concatenation schemes.

Chapter 4

Entropy Encoding in H.264

The H.264 advanced video coding standard for generic audiovisual services is defined in [1]. It was developed by the International Telecommunications Union - Telecommunications Sector (ITU-T) Video Coding Experts Group (VCEG) with the ISO/IEC Motion Picture Experts Group (MPEG). The H.264 standard by ITU-T and the MPEG-4 AVC standard by the ISO/IEC are jointly maintained with identical technical content and known as H.264/AVC. This standard boasts good video quality at much lower bit rates than previous standards with a minimal increase in complexity.

To an even greater extent than VC-1, H.264/AVC has a large set of profiles with different capabilities tailored to a wide range of applications. This video encoder targets High Profile, designed for high-definition television broadcast and disc-storage applications.

The H.264/AVC standard supports two entropy coding standards - CABAC (context-based adaptive binary arithmetic coding) and CAVLC (context-based adaptive variable length coding). Since CABAC requires a significant amount of processing but provides a much higher coding efficiency, CAVLC is made available as a lower-complexity alternative to CABAC. In the context of this project, the goal is to achieve high performance at low supply voltages through parallelism, so it is reasonable to support the more efficient CABAC algorithm in order to meet the specifications. As CAVLC can be seen as an extension of the VLC design for VC-1, it will not be implemented in this encoder design. All syntax elements not encoded by CABAC are encoded by exponential-Golomb coding.

4.1 H.264 Video Standards Study

As described in Section 1.4, the CABAC encoding flow can be broken down into several major steps. These steps are illustrated in Figure 4-1 and further discussed below.

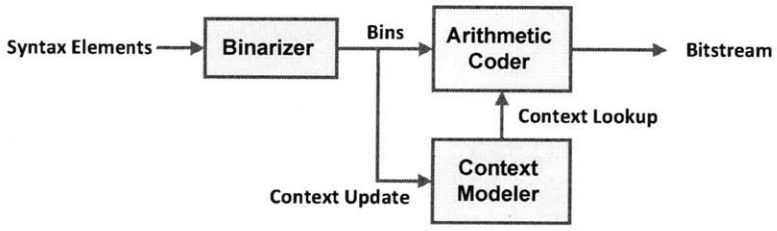

Figure 4-1: Major steps in CABAC encoding.

Since the encoding process illustrated in Figure 1-6 expects a binary input, the first step in the entropy coding engine of H.264 is to convert the non-binary units of video data (or syntax elements) into sequences of binary symbols. This process is generally referred to as binarization, and the binary symbols at its output are referred to as bins. The H.264 video standards document [1] contains detailed descriptions of the binarization process for each syntax element. Depending on the type of syntax element and other supplementary information such as frame type, the syntax element can be binarized using one or a concatenation of two variable length coding schemes. These include unary, truncated unary, exponential-Golomb, fixed-length, and table-based variable-length coding.

The binary arithmetic coder then processes the bins one by one. As shown in Figure 1-6, a sequence of three bins is encoded into a bitstream representation by three iterations of interval subdivision. For every given binary symbol, the arithmetic coder updates two internal state variables that completely characterize the current sub-interval of interest – these are the *range* (which indicates the length of the interval) and *low* (which indicate the lower bound of the interval) values. At each of these iterations, the probability model (or context) that dictates how the interval is to be subdivided can be updated. This can happen in one of two ways - if the type of syntax element to which the bin corresponds changes, then a context switch occurs; otherwise, if the context remains the same, the last bin encoded will cause an update to be made to the probability model. This adapting of

contexts based on the incoming data allows the CABAC encoder to learn from previous data statistics and increase the accuracy of the probability model.

The encoding process for a given bin can follow one of three flows. We will categorize each bin as being either a regular, bypass, or terminate bin, corresponding to the encoding flow that it follows. The regular encoding flow is the most common and general case – it corresponds to one step of the encoding process described in Section 1.4 and is illustrated in Figure 4-2. It takes the inputs *state* and *MPS* (most probable state) from the context modeler that determines the appropriate context to be used for the bin being processed. The *state* value indicates the probability that the value of the bin is equal to *MPS*, and *MPS* can either be a 1 or a 0 – the probability of the opposite bin value is simply complementary to the *MPS* probability. The value of the current bin is also used as an input. Based on these inputs, the *range* and *low* variables are updated for that bin. Firstly, an *rLPS* value is determined based on the current probability state and bits 7 and 6 (the second and third MSBs) of the previous *range* value. This value corresponds to the next range value given that the bin is an LPS (least probable symbol), and is an approximation of the multiplication $Range \times P(LPS)$. This mapping is defined in the standard as a 64x4 lookup table, corresponding to the 64 possible probability states and the 4 possible values of *range*[7:6]. The *rLPS* lookup corresponds to the multiplication between current *range* and probability of an *LPS* (least probable symbol), giving the location where the interval is to be divided, with the lower portion corresponding to the *MPS* and the upper portion corresponding to the *LPS*. If the value of the bin matches that of the *MPS* (i.e., the right branch in the flowchart is taken) then the *range* value is updated to the current range subtracted by the output of the table lookup (*rLPS*) and the *low* value remains the same, such that the lower portion is taken as the new interval. The *state* variable is also updated (by another lookup table) to indicate that the likelihood of encoding an *MPS* has increased due to an *MPS* being encoded in this stage. On the other hand, if the value of the bin does not equal that of the *MPS* (i.e. the bin is an *LPS*), the *range* value is set equal to *rLPS* and *low* is appropriately increased to indicate the new interval is the upper portion of the current interval. The *state* variable is updated (again according to a table defined in the standard) to effectively model the decreased likelihood of the *MPS* since the current bin being processed is an *LPS*.

As indicated in the Figure 4-2, an additional renormalization step checks that *range*

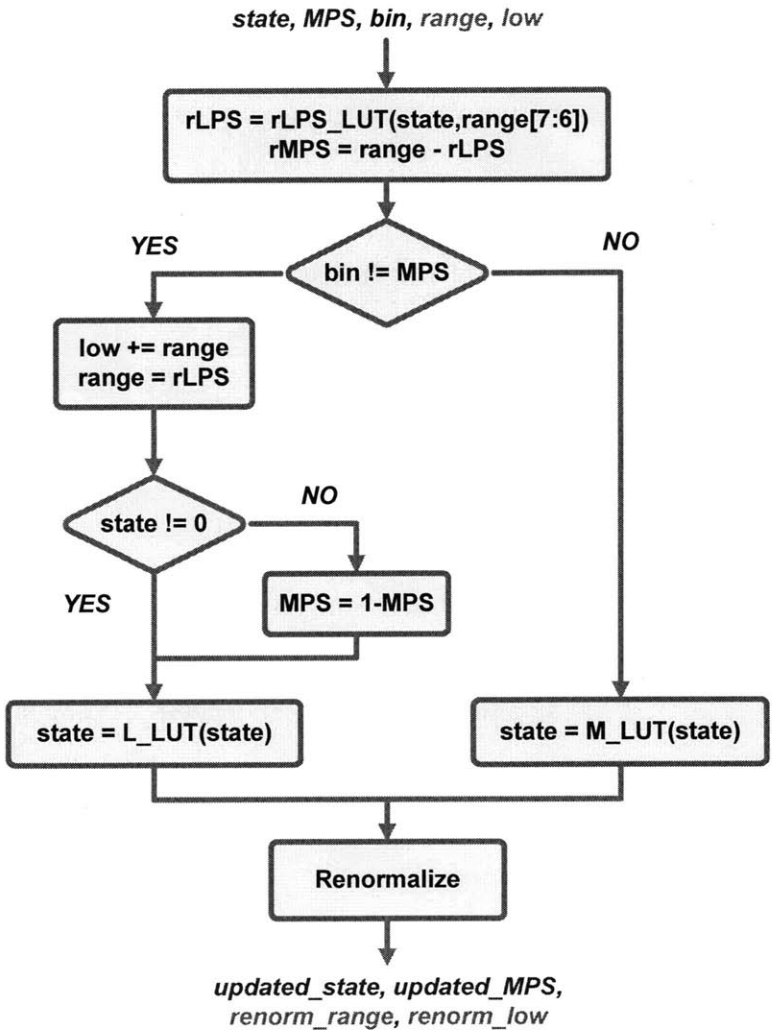

Figure 4-2: Regular binary encoding flow [1].

is above a minimum value – this check (and left-shifting for renormalization) is necessary because *range* and *low* are represented by a fixed number of bits and the multiplications and state transitions above are implemented by table lookup, which relies on *range* and *low* within a certain acceptable range of values. This process is described separately in Figure 4-3 and handles what bit is sent to the bitstream depending on the *low* value. Depending on the value of *low* during each iteration through the loop, one of three branches can be taken – if *low* is less than 256, a '0' is sent to the bitstream; if *low* is greater than 511, a '1' is sent; and for all values in between, an additional outstanding bit is added to the state of the engine by incrementing the internal counter called *bitsOutstanding*. Outstanding bits are accumulated by incrementing this counter until the next '0' or '1' is encountered, at which point the opposite bit is sent to the output a number of times equal to the outstanding bits counter. For example, if the encoder has accumulated 3 outstanding bits, the sequence *"1000"* is sent to the output if the next bit is a 1 and the sequence *"0111"* is sent if it is a 0. This process is described in Figure 4-4.

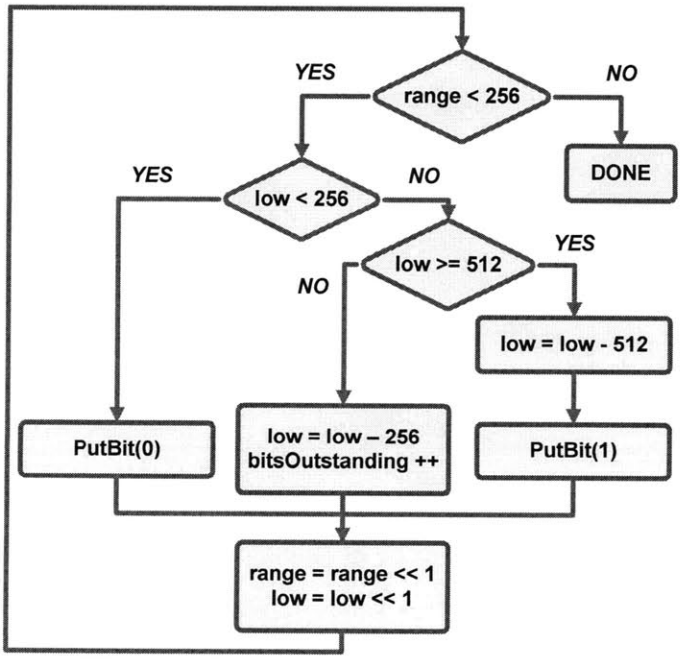

Figure 4-3: Renormalization [1].

The bypass encoding flow, as illustrated in Figure 4-5, is a simpler procedure that does not take *state* and *MPS* as inputs. This is because this flow assumes that a 0 or a 1 occurs

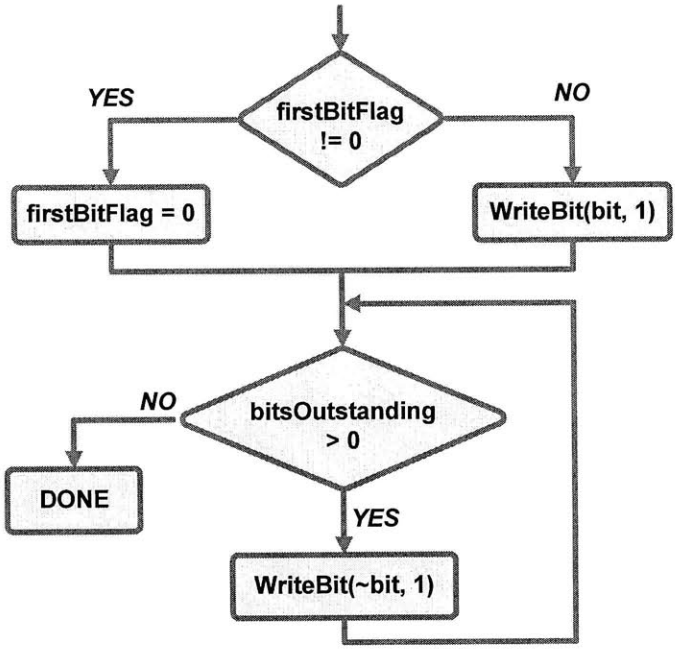

Figure 4-4: Output and outstanding bits handling [1].

with equal probability. This encoding flow is used for binary symbols with near-uniform probability distributions. It should also be noted that the bypass encoding flow incorporates renormalization instead of including the separate renormalization flow as a last step. The *low* value is firstly left-shifted by 1 bit regardless of the value of the bin. Then *low* is increased by the *range* from the previous iteration (i.e. half of the interval) if the bin is a 1 and unchanged otherwise – in other words, the lower half of the interval is taken to represent a 0 and the upper half corresponds to a 1. The bit sent to the bitstream depends on this new value of *low*, and *low* is updated again as per the branch taken in the decision tree.

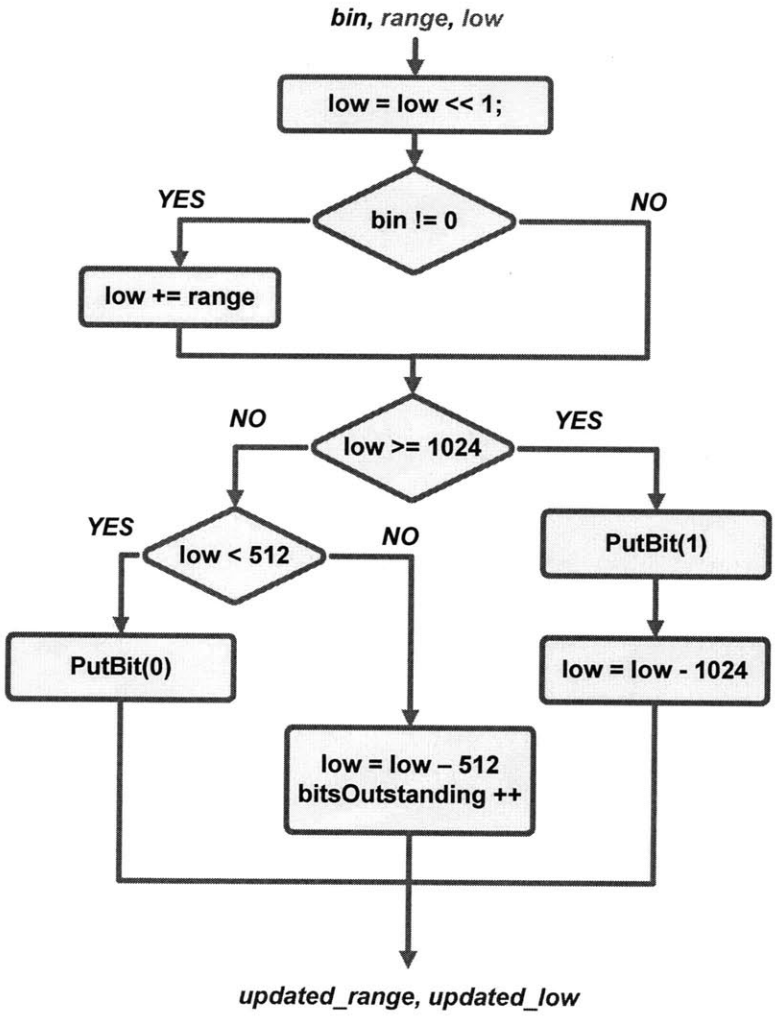

Figure 4-5: Bypass binary encoding flow [1].

Similarly, the terminate encoding flow, which is shown in Figure 4-6, only takes the current bin as its input and updates *range* and *low* according to its value. If the bin is a 0, *range* is decremented by 2 and the renormalization flow described by Figure 4-3 is used. Otherwise, *low* is updated by adding *range* – 2 and *range* is set to 2, after which renormalization is performed and the remaining bits are flushed and sent to the bitstream. This includes the handling of any outstanding bits that have accumulated from previous encoding iterations.

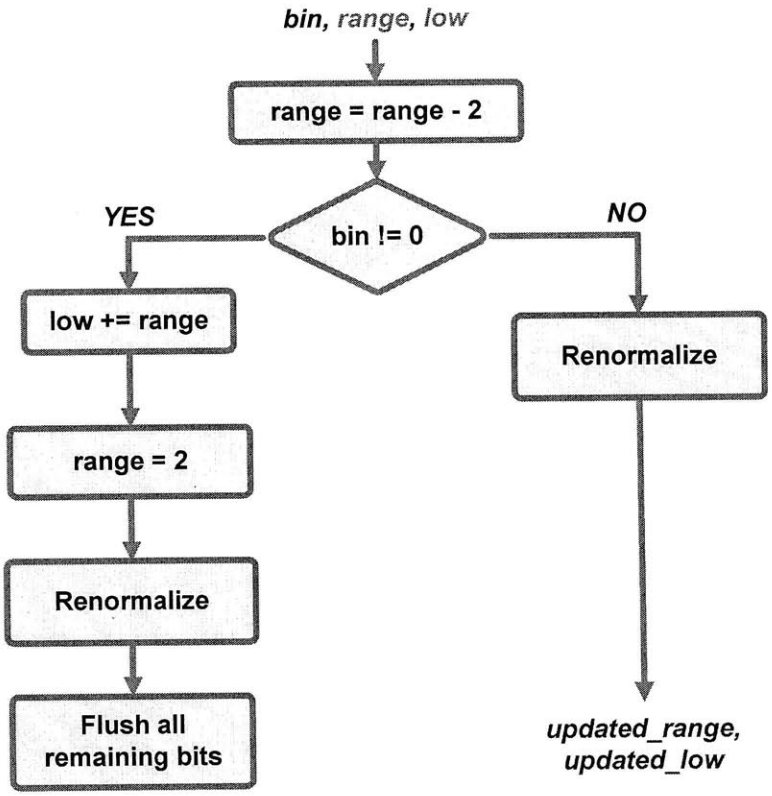

Figure 4-6: Terminate binary encoding flow [1].

The regular encoding flow described above relies on an accurate probability model. hlThe H.264 video standard defines 468 contexts, and each bin selects a particular context depending on a defined set of criteria. The distribution of these contexts among the syntax elements defined in H.264 is summarized in Table 4.1. For each context, the probability model is described by the *state* and *MPS* parameters, which are read and updated by the regular encoding flow. The update (or state transition) is made based on whether the bin encoded is an MPS or LPS. The *state* variable can take on values 0 to 63, where *state*=0 indicates a 50% probability that the next bin is an LPS, and increasing values of *state* indicate increasing probability of an MPS (and correspondingly decreasing probability of an LPS). Thus, the state transition for an MPS is involves incrementing the *state* variable by 1 every time an MPS is encoded, until the maximum value of 63 is reached at which point *state* saturates. Conversely, each time an LPS is encoded, *state* is decremented by different amounts depending on its current value. For example, when *state*=63, the next LPS encoded results in a large jump to *state*=38, and as *state* decreases the jump to the

next *state* given an LPS decreases as well. When *state*=0 and another LPS is encountered, the *MPS* bin toggles to indicate the probability of the former MPS has decreased past 50% and so is now referred to the LPS.

Table 4.1: Number of contexts assigned to each syntax element.

Syntax Element	**Number of Contexts**
mb_skip_flag	6
mb_field_decoding_flag	3
mb_type	24
transform_size_8x8_flag	3
coded_block_pattern (luma)	4
coded_block_pattern (chroma)	8
mb_qp_delta	4
prev_intra4x4_pred_mode_flag, prev_intra8x8_pred_mode_flag	1
rem_intra4x4_pred_mode, rem_intra8x8_pred_mode	1
intra_chroma_pred_mode	4
ref_idx_l0, ref_idx_l1	6
mvd_l0[][][0], mvd_l1[][][0]	7
mvd_l0[][][1], mvd_l1[][][1]	7
sub_mb_type[]	7
coded_block_flag	32
significant_coeff_flag[]	152
last_significant_coeff_flag[]	140
coeff_abs_level_minus1[]	59
Total	**468**

4.2 Specifications

Since the CABAC operation is tied to the bin rather than the syntax element (i.e., one bin is encoded per step of interval subdivision and renormalization), it is constructive to determine the number of bins that need to be encoded per cycle when specifying the requirements for the entropy encoder module. From simulating the "duckstakeoff" 1080p standard

HD sequence, which has the worst-case bitrate of 35093.28 kbits/second at a frame rate of 50 frames/second, a peak value of 1,779,605 bins are encoded in each frame, and the distribution of the number of bins per frame for 10 frames is plotted in Figure 4-7. The peak value is mapped to 7,118,420 bins encoded per 4Kx2K frame under the assumption that the number of bins in a frame scales approximately linearly with the number of pixels in the frame (a 4Kx2K frame has roughly four times as many pixels as a 1080p frame). Assuming three frames are encoded in parallel at at target frequency of 25 MHz at low voltage, the targeted number of bins to be encoded per cycle in each parallel engine can be computed as follows:

$$\begin{aligned} \text{NumBinsPerCycle} &= \lceil \frac{(7,118,420 bins/frame) \times (50 fps)}{(25 MHz) \times (3 engines)} \rceil \\ &= 5 \text{bins/cycle per engine} \end{aligned} \tag{4.1}$$

The target is set at 6 bins to give a safety margin needed in case of irregular sequences or higher peak values. The H.264 entropy encoder engine must support the maximum bit rate specified in the standards document [1].

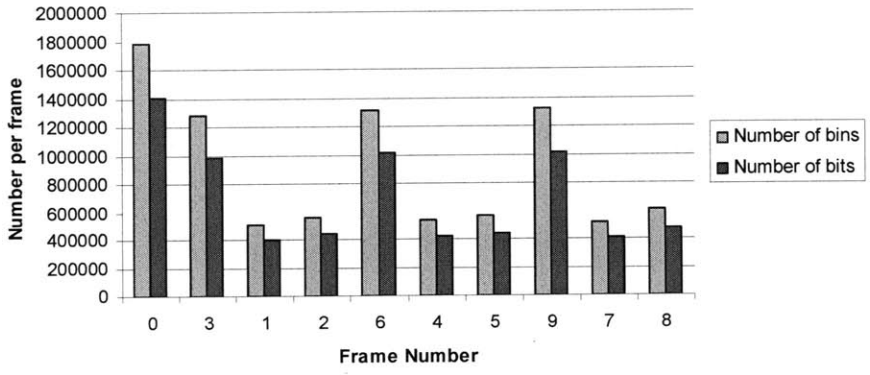

Figure 4-7: Number of bins per 1080p frame for the *Duckstakeoff* sequence.

4.3 Simulation with Reference Software

The H.264 reference software was used to determine the bin processing requirement that fixes the specifications for the H.264 entropy encoder module. This was done by inserted

a counter in the code that gets incremented appropriately each time bins are sent to the arithmetic coding function. To map the maximum bit-rate defined in the standards [1], this bin count value is compared to the number of bits written when encoding several standard HD video sequences to obtain an approximate ratio of number of bins per bit. This was found to be 1.3 bins for every bit sent to the bitstream, which is consistent with the 4 bins per 3 output bits ratio provided in the standard [1].

Chapter 5

Entropy Encoder Architecture for H.264

The H.264 entropy encoder in this design implements the highly serial CABAC (Context Adaptive Binary Arithmetic Coding) in hardware. There are two major sources of serial dependency inherent to the algorithm: context-adaptivity and interval subdivision. The former refers to the need to update probability models (referred to as contexts) because probability estimates are updated as more data is encoded such that the context can more closely model the actual data statistics. The latter describes the relationship between the *range* and *low* information of current and subsequent bins.

To meet the throughput requirement specified in Section 4.2, the entropy encoder module for H.264 is designed to encode six bins in each cycle. Unlike the architecture design for the VC-1 entropy encoder, it is not possible to completely parallelize the encoding operations of consecutive binary symbols (bins). Each step of interval subdivision (as described in Section 1.4) relies on the *range* and *low* values from the previous stage, so encoding multiple bins per cycle is only possible through the use of a cascaded structure. The generation of the next *range* and *low* values based on values from the previous bin forms a critical timing path that increases with each additional bin processed in a single cycle. This chapter discusses a pipelined architecture that provides the 6-bin requirement by dividing the CABAC operation into four stages.

5.1 Architecture Design

As discussed above, a four-stage pipelined architecture is used in this design. Figure 5-1 illustrates the pipeline partitions and the data that is passed from each stage to the next. A detailed description of each pipeline stage follows.

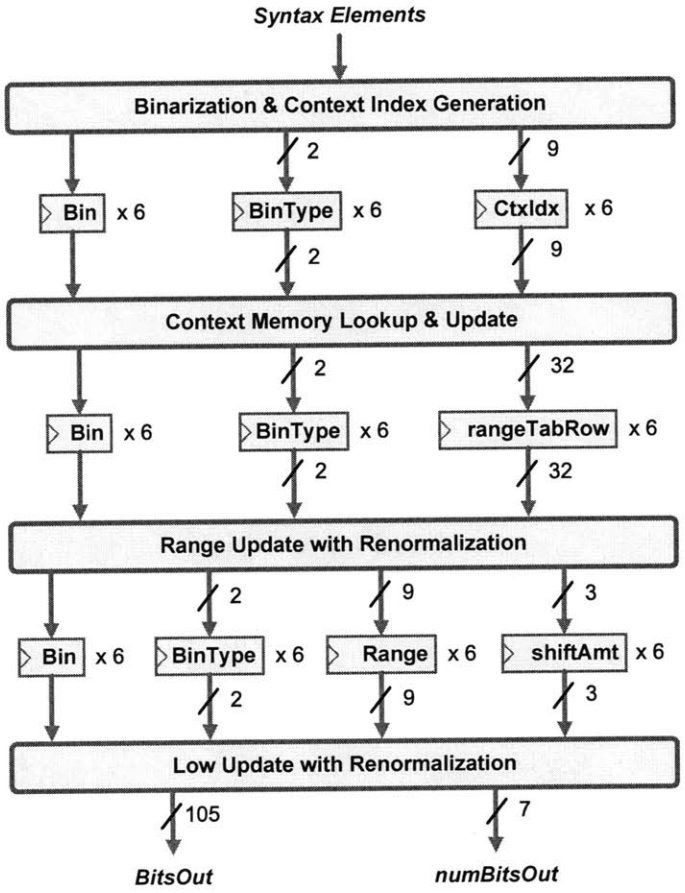

Figure 5-1: Pipeline Architecture for H.264 Entropy Encoder.

5.1.1 Binarization and Context Index Generation

Binarization describes the mapping of non-binary syntax elements to a binary representation (sequence of bins), which is sent to the arithmetic encoder to process using the CABAC flow, as described in Section 4.1.

The binarization process also determines whether a bin should be encoded using the regular, bypass, or terminate encoding flow. This is determined by the type of syntax element as well as the bin position in the binarized form of the syntax element and neighbouring

block information. For example, the syntax element /emphmb_type (macroblock type) for I slices is binarized according to a table defined in the standard, partially shown in Table 5.1. Each binIdx holds a '1' or a '0', and the probability models of the bins can be different for different values of binIdx. This is because the bins are not all equally probable to have one value versus the other, and so are modeled by different contexts. By the same argument, a different encoding flow can be selected based on the value of binIdx. Context index generation should be done at the same time since each bin at the output of binarization can belong to a different context and so context indices are also tied to the bin rather than the syntax element. In the encoding flow, the binarization step is independent from the rest of the entropy encoder in the sense that there are no feedback paths from the context modeling (updating contexts after encoding each bin) and arithmetic coding (interval subdivision and renormalization) stages, as opposed to the decoder case where binarization is part of a feedback loop. The same holds true for context index generation since the model selected to estimate the probability statistics of any given bin is fully characterized by the syntax element to which the bin belongs and the bin's position.

Table 5.1: Binarization (partial) for macroblock types in I slices [1].

Value of mb_type	**Bin String (binIdx)**						
	0	1	2	3	4	5	6
0 (I_NxN)	0						
1 (I_16x16_0_0_0)	1	0	0	0	0	0	
2 (I_16x16_1_0_0)	1	0	0	0	0	1	
3 (I_16x16_2_0_0)	1	0	0	0	1	0	
4 (I_16x16_3_0_0)	1	0	0	0	1	1	
5 (I_16x16_0_1_0)	1	0	0	1	0	0	0
6 (I_16x16_1_1_0)	1	0	0	1	0	0	1
7 (I_16x16_2_1_0)	1	0	0	1	0	1	0
8 (I_16x16_3_1_0)	1	0	0	1	0	1	1
9 (I_16x16_0_2_0)	1	0	0	1	1	0	0
10 (I_16x16_1_2_0)	1	0	0	1	1	0	1
11 (I_16x16_2_2_0)	1	0	0	1	1	1	0
12 (I_16x16_3_2_0)	1	0	0	1	1	1	1
13 (I_16x16_0_0_1)	1	0	1	0	0	0	
14 (I_16x16_1_0_1)	1	0	1	0	0	1	
15 (I_16x16_2_0_1)	1	0	1	0	1	0	
16 (I_16x16_3_0_1)	1	0	1	0	1	1	

Since the binarization and context index generation block operates on individual syntax

elements and can be parallelized as necessary to generate six bins per cycle, it is not the bottleneck of the entropy coding flow in H.264 and thus its implementation is not discussed here. For the purposes of the other pipeline stages downstream to this stage, it is assumed to produce six sets of bin values ('0' or '1'), context indices, and bin type (regular, bypass, or terminate), one for each bin processed in the same cycle.

5.1.2 Context Memory Lookup and Update

The context memory is the hardware realization of the probability models that dictate the encoding flow of each bin. For each of the 468 context models defined for H.264 High Profile with 4:2:0 colour format, a 6-bit value for probability stage (*pState*) and a 1-bit value indicating the most probable symbol (*MPS*) with probability corresponding to *pState* is maintained in the context memory. The groups of context indices (tied to the syntax element containing bins that use these probability models) are tabulated in Table 4.1. Here this memory is implemented as a register file to allow the flexibility for up to six different context values to be queried in parallel (assuming the case that all six bins being processed in a given cycle belong to different context models). Given the six context indices from the previous pipeline stage, the context memory is queried for the appropriate context models, and once a bin is encoded, an updated *pState* is written back to the memory to adapt the probability models to actual data statistics. This process is repeated every cycle until a terminate bin is reached, indicating the end of the slice. At that point the bitstream is flushed and the context memory needs to be re-initialized as defined in the standards document [1].

Context Memory Initialization

The initial values of *pState* and *MPS* (stored in read-only memory) for each of the context models are defined in the standards in terms of two variables, m and n. These variables, along with the quantization for the current slice ($SliceQPY$), are used to compute the probability state (*pStateIdx*) and most probable symbol (*valMPS*) according to the following pseudo-code:

```
preCtxState = Clip3(1, 126, (( m * Clip3(0, 51, SliceQPY)) >> 4) + n)
if( preCtxState <= 63 ) {
```

```
    pStateIdx = 63 - preCtxState
    valMPS = 0
} else {
    pStateIdx = preCtxState - 64
    valMPS = 1
}
```

The Clip3(x,y,z) function returns the value of z as long as it lies between x and y; otherwise, it returns the boundary value which it exceeds, thus saturating the value of z to the range provided by x and y. Since the values m and n are variables dependent on slice type, and m is one of the inputs to a multiplier, the appropriate m value is multiplexed and the output of the multiplexer is used as a fixed input to the multiplier. Given the value of SliceQPY, all initialization values are computed in parallel to avoid the control complexity of interfacing between multi-cycle context initialization and the other modules in the video encoder.

Context Memory Lookup

Since the six bins being encoded in parallel can refer to the same context models, and the correct behaviour is for any previously encoded bins to update the context model before a later bin referring to the same context index uses it for interval subdivision. Thus, the context lookup should be implemented such that later bins check previous bins for updated context values before querying the potentially stale value in the context memory. This data hazard can be classified as a read-after-write (RAW) hazard, and can be avoided by additional comparison logic that checks whether the value in the context memory is updated. For example, for the sixth concurrently encoded bin in a given cycle, a comparison needs to be made with each of the five previous bins on the context index corresponding to each bin. This comparison is made in order with the highest priority given to the fifth bin, then the fourth, and so on, since the most recently encoded bin with the same context index provides the updated value for the sixth bin. In the worst case where all six bins refer to a single context model, the updated *pState* and *MPS* values from the first bin are used as the result of the context lookup for the second bin, and so on, thus forming a critical path from cascading all six processing units, shown schematically in Figure 5-2. Note that the figure

only shows the context index comparison with the immediately preceding bin (for simplicity and sufficient for demonstration of worst case critical path where all six bins use the same context model); additional comparators and multiplexers are used at each bin to check the context indices of all preceding bins and select the correct bypass value accordingly in the actual implementation. The length of this path depends on the updating mechanism of the context model, which is discussed next.

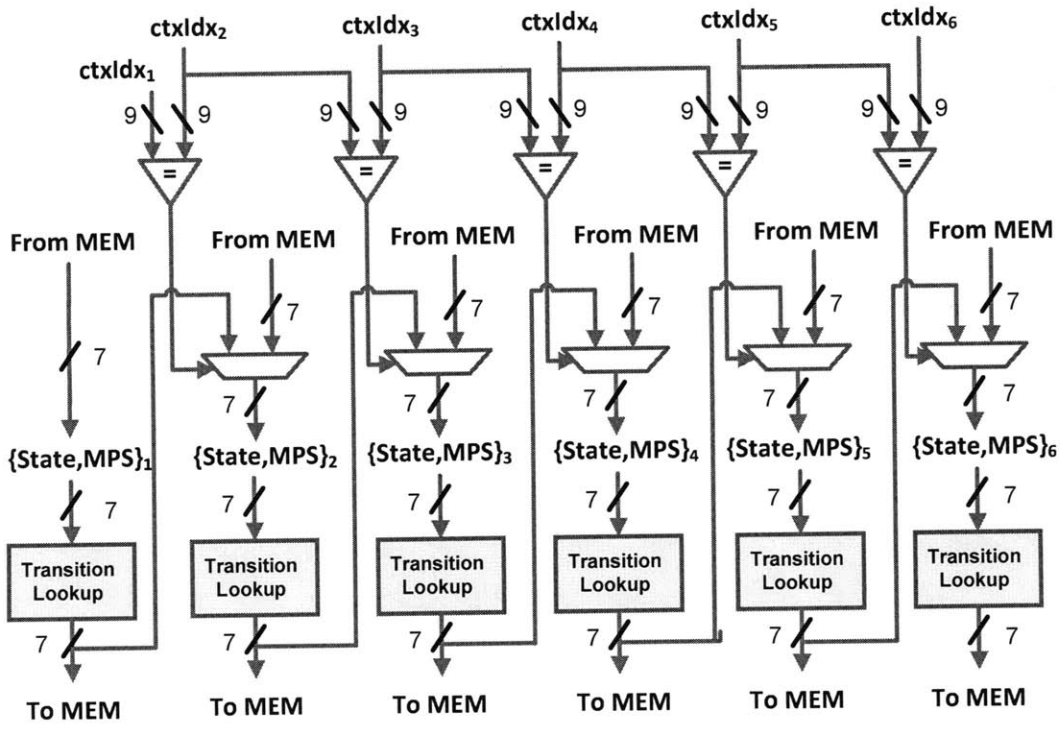

Figure 5-2: Cascaded structure of 6-bin context lookup stage.

Context Memory Update

Given a value of *pState* and *MPS* from the context lookup, the current bin value determines the transition in *pState* and whether the *MPS* value is toggled. The probability state is defined as a 6-bit *pState* value, and once the bin to be encoded is determined to be an MPS or LPS, the corresponding state transition table is consulted for the new *pState* value. Since these tables is fixed, the lookup has the equivalent delay of a 64-to-1 multiplexer. Then, the new state is written back to the context memory to update the probability model. The new *pState* is also checked to see if the lowest value 0 has been reached. At this boundary, the probability of the current *MPS* value has dropped past 50%, which means it

is no longer the most probable symbol for this context – in this case, the bit is flipped to indicate a change of MPS, and this also causes a writeback to context memory.

5.1.3 *Range* Update with Renormalization

The *range* value, as discussed in Section 4.1, is updated each time a bin is processed, and is held as an internal variable in the arithmetic encoder. Since the *range* value for the current bin is based on the *range* value from the previous bin, this process is highly serial and cannot be completely parallelized. In order to process multiple bins per cycle, the processing unit that generates each incremental *range* value must be placed in a cascade, and the critical path of this stage increases with the number of bins.

In order to meet the timing constraint of 20 ns per cycle (corresponding to an operating frequency of 50 MHz), circuit level optimizations must be used to reduce the delay through each additional bin to be processed. The implementation must also handle the three types of encoding flows, namely *regular*, *bypass*, and *terminate*, discussed in Section 4.1. Thus, at each intermediate stage in the 6-bin cascaded structure, each representing the processing unit of a single bin, the updated *range* value from all three encoding flows is computed, and a 3-to-1 multiplexer is used with a 2-bit select input generated by the binarization stage. This path is shown in Figure 5-3. The overall critical path is analyzed after a discussion of each of the *regular*, *bypass*, and *terminate* processes for *range* update.

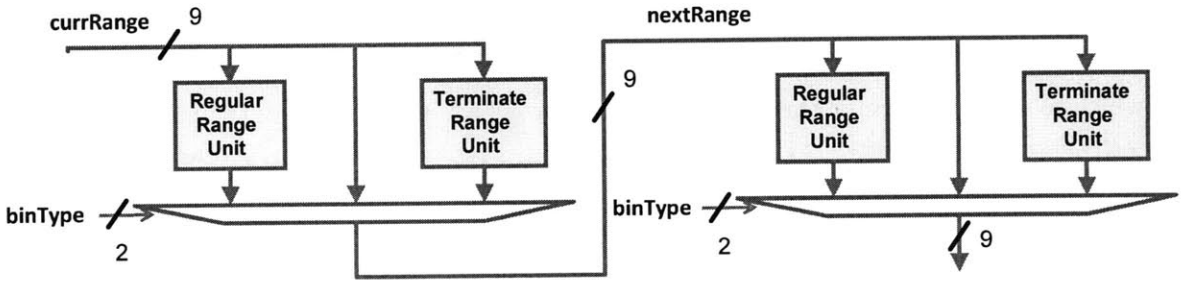

Figure 5-3: Cascaded structure of 6-bin *range* update stage.

Range Update for Regular Bins

From the regular encoding flow illustrated in Figure 4-2, we can identify the steps involved in computing a new *range* value. The *rLPS* value, corresponding to the length of the interval that represents the least probable symbol in the current probability model, is

firstly determined by a 64x4-entry table lookup operation. In the *rangeTabLPS*, each row of four *rLPS* values corresponds to one of the 64 possible values of *pState*, and each of the four columns corresponds to a two-bit value of the bits *range*[7:6]. The lookup can be viewed as a two-step process where a 64:1 row selection is performed before a 4:1 lookup on the selected row is made. The *rLPS* value from this lookup is then subtracted from the previous *range* value to produce an intermediate value for the new *range*; this intermediate value is either taken as the new *range* (to be renormalized) if the bin being encoded is the least probable symbol according to the probability model or overwritten with the *rLPS* value if an MPS (most probable symbol) is detected. In hardware, this is implemented using an adder for the subtraction and a multiplexer to select the *rLPS* value or the result of the subtraction, depending whether the bin is an MPS or LPS. The output of the multiplexer then undergoes the renormalization process illustrated in Figure 4-3. Since the software flow iteratively performs 1-bit left-shifting operations to the *range* value until it exceeds the value 256, and the *range* value is implemented with the minimum required precision of 9 bits, the number of iterations for which this loop is executed is equal to the number of leading zeros in the *range* value prior to renormalization. For example, if *range* equals 95, its binary representation is *9'b001011111*, so the loop is entered twice, once to shift *9'b001011111* to *9'b010111110* or 190, and a second time to shift *9'b010111110* to *9'b101111100* or 380, which exceeds 256, at which point we exit from the renormalization loop. This behaviour can be produced using a leading zero detection followed by a barrel shifter. The leading zero detector takes the *range* value as an input and computes the number of leading zeros, which is used to specify the number of bits by which the barrel shifter should shift the *range* value to the left to obtained the renormalized *range* (to be passed to the next interval subdivision stage).

Thus, for each of the cascaded stages, the inputs are the *range* value from the previous stage (or the previous cycle in case of the first bin in the current cycle) and the probability model *pState*. These values are used to to select the appropriate entry in the *rangeTabLPS*, which is then used in the computation for the new *range* value. This means that the critical timing path in a single stage passes through an 8-bit-wide, 64x4-to-1 table lookup, a 9-bit adder, a 2-to-1 multiplexer, a leading zero detector, and a 9-bit-wide barrel shifter. This path is delineated by the arrow in Figure 5-4.

For a 6-bin cascaded structure, since the table lookup at each stage requires the output

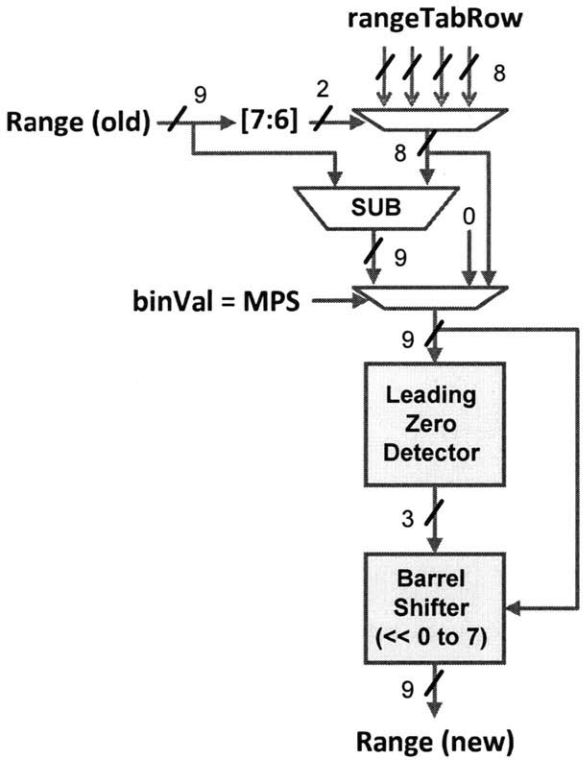

Figure 5-4: Critical timing path in a single stage of *range* update and renormalization.

of the previous stage, the overall critical timing path is approximately six times as long as the single-stage critical path. The only other pipeline stage that has comparable degrees of dependency from stage to stage is the *low* update (to be discussed in Section 5.1.4), but the critical path of each stage in *low* update is significantly shorter because no table lookup is needed. Hence, the *range* update is expected to be the most time-critical of the pipeline stages, and additional optimizations are explored to reduce the length of this critical path so that the targeted clock frequency can be met. This optimizations are discussed in Section 5.2.

Range Update for Bypass Bins

As the bypass encoding flow shown in Figure 4-5 does not make any changes to the *range* value, when a bypass bin is encountered, the *range* value is directly passed through to the next stage.

Range Update for Terminate Bins

When encoding a terminate bin (when the bin equals 0), the *range* value is used to flush out all outstanding bits stored from previously encoded bins. This is achieved by setting *range* to 2, which then causes the renormalization loop to be executed seven times since *9'b000000010* has seven leading zeros. If the bin equals 1, it is simply decremented by 2 and renormalized as in the case of a regular bin.

5.1.4 *Low* Update with Renormalization

Since the *low* value depends on the *range* value both for interval subdivision and renormalization, both the *range* value itself and the number of bits by which *range* was left-shifted during renormalization (henceforth referred to as the *shiftAmt*) of each of the six bins is passed to the *low* update pipeline stage through registers. The type of encoding flow for each bin is also passed from the *range* update pipeline stage through a 2-bit register to indicate whether it is a regular, bypass, or terminate bin. As in *range* update pipeline stage, maximum flexibility is achieved by computing the updated *low* value from all three flows and selecting the appropriate one, and six of these multiplexed structures are cascaded as shown in Figure 5-5. The hardware implementation for updating and renormalizing the *low* value in each of these flows is discussed below.

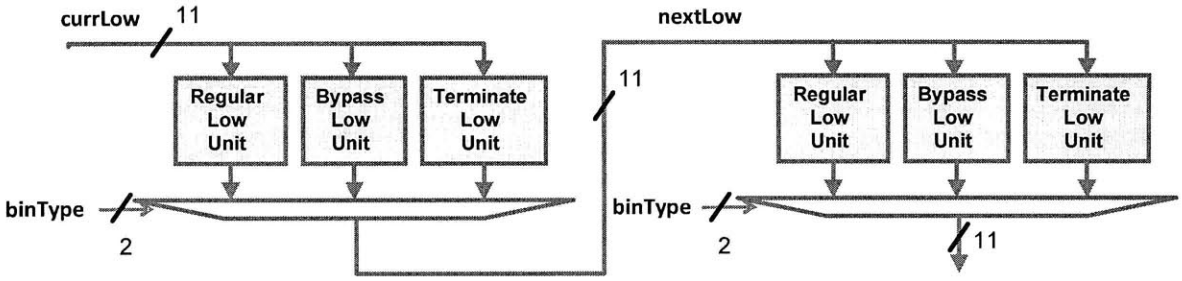

Figure 5-5: Cascaded structure of 6-bin *low* update stage.

Low Update for Regular Bins

In the regular encoding flow shown in Figure 4-2, the *low* value is either maintained (when the bin is the most probable symbol and thus the lower segment of the interval is taken) or increased by *range* (when the bin is the least probable symbol and the upper

segment is taken). This intermediate *low* value is then subjected to the same renormalization loop as the *range*; however, since the number of iterations for which this loop is executed is completely determined by the number of leading zeros in *range* prior to renormalization, the *shiftAmt* value provided by the previous pipeline stage can be used directly by the barrel shifter delivering the renormalized value of *low*. This process is relatively simple and the barrel shifter can be implemented the same way as the barrel shifter used for *range* renormalization.

The complexity of the *low* update pipeline stage comes from the fact that one bit is generated in each iteration of the loop. Since the number of bits sent to the bitstream depends on both the number of iterations through the renormalization loop (i.e., on the value of *shiftAmt*) and the intermediate value of *low* during each iteration, the hardware implementation needs to unravel the loop and a unique sequence of output bits is added to the output bitstream depending on the ordered set of branches taken. As expected, the higher the value of *shiftAmt*, the more cases must be exhaustively considered and handled. For example, if *shiftAmt* equals 7, there are 4×2^7 possible sets of output bits and values of the *bitsOutstanding* counter to be considered; the factor of 4 comes from the fact that the two MSB bits of the pre-renormalization value of *low* results in different branches taken in the first two iterations through the loop, and for each of these ranges of *low* values a different set of 2^7 cases are possible. The resolution of outstanding bits also causes each case to be unique since the actual value sent to the output depends on the first non-outstanding bit received.

Within the block of six bins being processed per cycle, the value of *bitsOutstanding* at the output of the processing unit for one bin is fed into that of the next bin as an input, and so on. While some outstanding bits may be resolved by the subsequent bins within the same cycle, it cannot be assumed that the sixth bin processed in each cycle will not add to the *bitsOutstanding* counter, as each bin behaves in the same way, independent of the six-bin cycle boundary imposed by the hardware structure. This is handled through the use of an internal register which holds the value of *bitsOustanding* at the end of the cycle, and feeds it as an input to the first bin in the next cycle. The *low* value at the end of each cycle is similarly registered and provided as an input at the beginning of the next cycle.

The critical timing path of the *low* update stage for a single bin is illustrated in Figure 5-6. It can be seen that this path must be shorter than the critical timing path of the *range*

update stage shown in Figure 5-3, which means the overall encoder critical path is set by the *range* update stage.

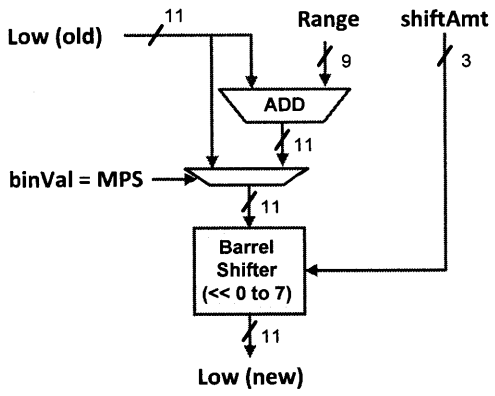

Figure 5-6: Critical timing path in a single stage of *low* update and renormalization.

Low Update for Bypass Bins

The *low* value for bypass bins is computed through operations similar to one iteration of the renormalization loop in the regular bin case. The bypass encoding flow essentially assumes the values '1' and '0' are equiprobable, such that the current interval is divided in half and the lower or upper segment is taken as the new interval. The equal division of the interval is achieved maintaining the previous *range* value and doubling the *low* value. This provides the advantage that no renormalization is necessary since the *range* value has not changed and no additional leading zeros are introduced (the *range* value remains > 256). Then the value of *low* is maintained if the bin is a '0' and increased by *range* if the bin is a '1'. This value of *low* then determines the bit sent to the output bitstream (or whether *bitsOutstanding* is incremented). To correct for the fact that the *low* value was doubled at the beginning of the flow, the thresholds to which *low* is compared for each branch are also doubled (i.e., if $low < 512$, a '0' is sent; if $low > 1023$, a '1' is sent; otherwise, *bitsOutstanding* is incremented by 1). This is implemented in a similar fashion to the *shiftAmt* equals 1 case, which guarantees that at most one bit is sent to the output bitstream.

Low Update for Terminate Bins

The terminate encoding flow either maintains the *low* value or increases it by *range* depending on the value of the bin being encoded (similar to the bypass case). The *low*

value then undergoes renormalization as in the case of a regular bin, so the barrel shifter module described above is shared by the terminate and regular bin encoding units.

5.2 Optimizations for Critical Path Reduction

As argued in Section 5.1.3, the *range* update stage has the longest critical path out of all the stages in the pipeline architecture. Since the video encoder chip is targeted to operate at 25 MHz with a supply voltage of 0.6V, and synthesis is performed at 0.81V, the maximum critical path length is obtained through experiments that indicate the approximate scaling of frequency with decreasing V_{DD} down to 0.6V. This calculation mapped 25 MHz at 0.6V to approximately 43.5 MHz at 0.81V, which translates to a maximum critical path length of 23 ns. With the *range* update pipeline stage implemented as in Figure 5-4, initial synthesis results indicate that the critical timing path is 28 ns, which exceeds our calculated maximum. Leaving a safety margin of 15%, three architectural optimizations are made in order to meet the target clock period of 20 ns.

To minimize the length of the critical path, the 64x4-to-1 *rangeTabLPS* table lookup is decomposed into two multiplexer stages as described in Section 5.1.3. Instead of taking the *pState* value from the context lookup pipeline stage and performing both steps of the table lookup in the *range* update stage, the 64-to-1 lookup can be done in the context lookup stage and its output (corresponding to the selected row in the *rangeTabLPS* table) can be fed into the pipeline registers and supplied to the *range* lookup stage. This reduces the table lookup delay down to a 4-to-1 multiplexer delay.

The delay through each component along the critical path is limited by the last arriving input to that block. Since the six bins are processed in a cascaded fashion and the updated *range* value of one bin is used as an input of the next, the 9-bit *range* signal is always the last available input. The 4-to-1 multiplexer takes bits 7 and 6 of the previous *range* value as a selection input, and this limits the rest of the datapath to wait for *range* to arrive. The subtraction of *rLPS* from the previous *range* value also requires this last arriving input, before the new range value can be selected based on whether the bin is an MPS. After this 2-to-1 selection, the pre-renormalization *range* value passes through the leading zero detector to determine the number of bits by which to left-shift *range* for renormaliziation (i.e., compute *shiftAmt*, which is an input to the barrel shifter).

The shifting operation can be performed in parallel with the leading zero detection if the *shiftAmt* value is not required before shifting can occur. This can be achieved by replacing the single barrel shifter with multiple shifters that assume different values of *shiftAmt*, and selecting the appropriate output using the *shiftAmt* output from the leading zero detector. The *shiftAmt* value can range from 0 to 7 depending on the value of *range*, so the original combined delay of the leading zero detector followed by the barrel shifter is reduced to that of the leading zero detector followed by an 8-to-1 multiplexer. This structure is illustrated in Figure 5-7. Since the 8 shifters simply left-shift the input by a fixed number of bit positions, they should not be too costly in terms of area consumption. Synthesis results summarized in Section 5.4 indicate that this technique reduces the critical timing path length from 27.38 ns to 20.05 ns while incurring the cost of 2.38K gates in the *range* update pipeline stage from pre-computing the output of the barrel shifter using 7 hard-coded shifters.

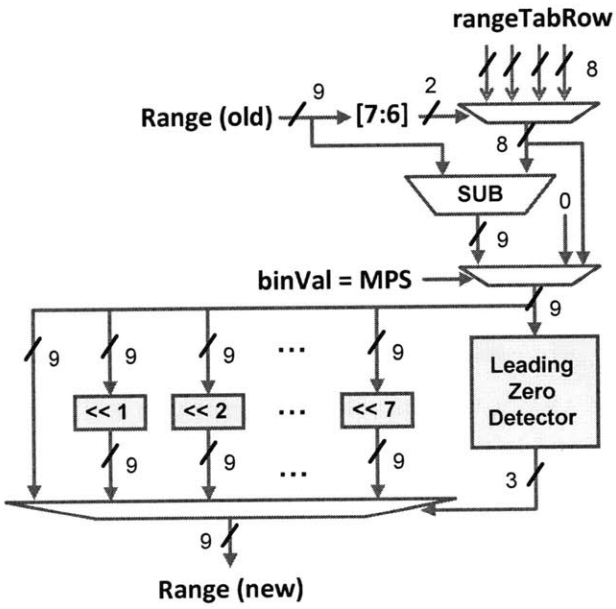

Figure 5-7: Reduced critical timing path in a single stage of *range* update and renormalization.

5.3 Verification

The CABAC core functionality performs a mapping from bins to an output bitstream. These bins are generated by the binarizer, which is removed from our implementation, and

thus bins serve as the input to this module. The test vectors are generated by inserting print statements at each point binarization is performed in the reference software. Test vector files contain a bin value ('0' or '1'), a context index (indicating which entry in the context memory should be used to encode that bin), and a bin type (regular, bypass, or terminate) for each bin generated by the software, and the testbench file generates a bitstream representation based on these inputs. The output bitstream is then compared to the bitstream generated by the reference software to verify functionality.

Figure 5-8 shows the waveforms generated from the *range* update pipeline stage which takes *encode_type* (regular, bypass, or terminate), *rangeTabRow* (four possible values for *rLPS*), *binVal* ('1' or '0') and *binIsLPS* (whether the bin being encoded is a least probable symbol according to the probability model) for each of the six bins as inputs and generates the *shiftAmt* (the number of times *range* is shifted is equal to that for *low*, values from 0 to 7) for each bin in the *low* renormalization stage as its outputs.

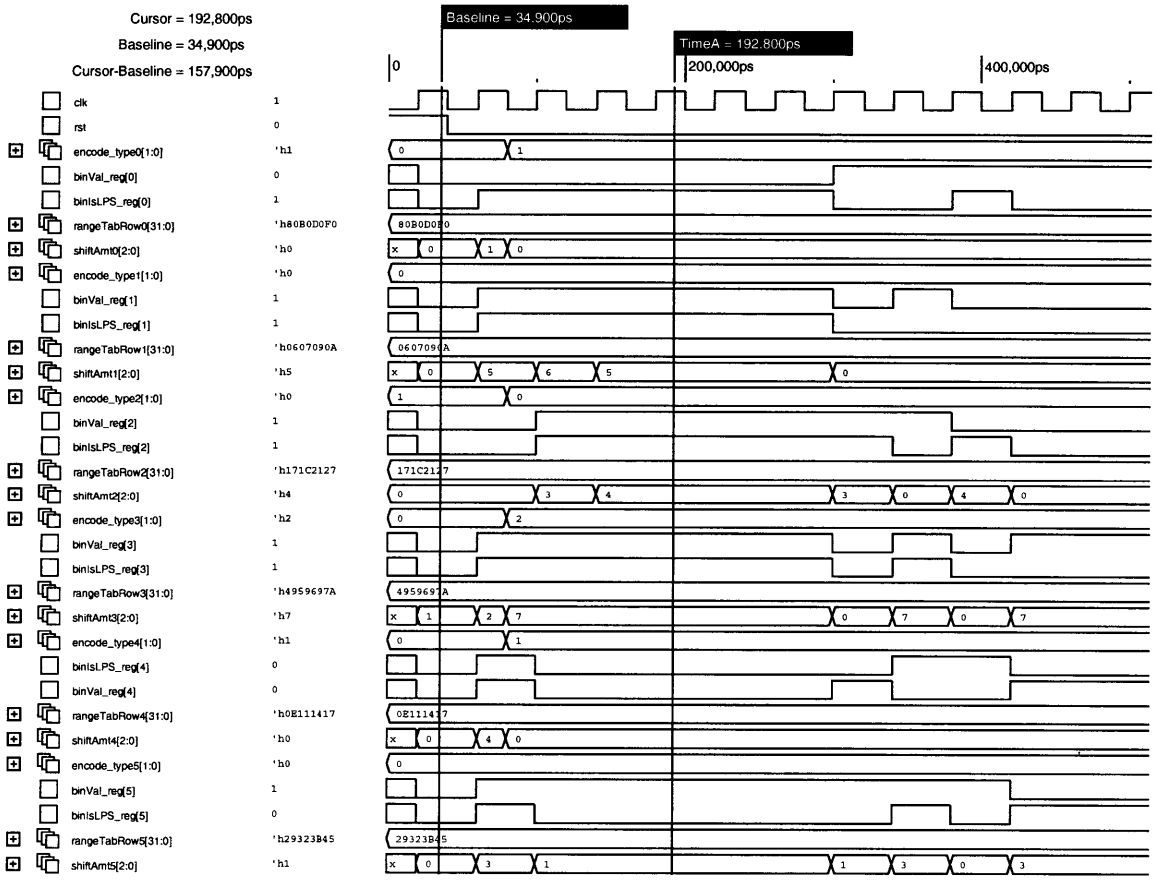

Figure 5-8: Waveform from verification of *range* update and renormalization pipeline stage.

5.4 Synthesis Results

The H.264 entropy encoder module was synthesized using a 65nm library in Synopsys Design Compiler. As in the case of VC-1, the critical path constraint was set to 5 ns (200 MHz), which approximately scales to 20 ns (50 MHz) at a supply voltage of 0.6V, in order to determine the minimum length of the critical path. The total gate count of the module is 411.3K gates, which was obtained by dividing the reported total cell area by the size of the smallest NAND gate as defined in the library. The achieved bin rate is discussed and compared with various previous works surveyed in Section 1.4.2. The critical path length prior to the optimizations discussed in Section 5.2 is 27.38 ns, which exceeds the targeted 23 ns. Thus, the optimized design is implemented (increasing the gate count to 452.6K) and its area overhead versus timing reduction tradeoff is discussed.

5.4.1 Bin-rate Achieved versus Previous Works

Table 5.2 summarizes the reported bin rate, operating frequency, and throughput of several previously published H.264 entropy encoders. In this implementation, the number of bins processed per cycle is guaranteed to be six, and the operating frequency is computed from the critical path delay at nominal voltage. For comparison with previous works, the bin rate calculation is made with frequency scaled to the equivalent of 276 MHz at nominal voltage. Given that three frames of video data will be processed in parallel, the overall throughput is found to be 4.97 Gbins/second, which is several orders of magnitude higher than some of the previous works. Note that this implementation also supports the H.264 High Profile as opposed to the Baseline and Main Profiles supported by most earlier works.

The throughput requirement of the entropy encoder is aggressive due to the need to support ultra high-definition encoding in real-time. Moreover, CABAC dominates in timing in the overall system due to its highly serial nature while other blocks like motion estimation dominates area due to the large amounts of SRAMs needed. Thus, the engineering trade-off taken in this work is different from that of a stand-alone entropy encoder, as presented in the previous works list; this work focusses on maximizing throughput at the expense on added area since this additional hardware cost is absorbed into the overall video encoder system. In spite of this fundamental difference in approach, it is still constructive to highlight some similarities and differences with some of these existing encoders.

Table 5.2: Throughput comparison with previous works.

Design	Bins Per Cycle	Frequency (MHz)	Throughput (Mbins/s)
[11]	0.06	100	6
[12]	0.588	150	80
[13]	1	200	200
[15]	0.56	333	186
[16]	0.67	200	134
[19]	1	130	130
[21]	1.9 to 2.3	186	353 to 427
[23]	0.33	263	86
[22]	1	362	362
This work	**6**	**276**	**1656**

In [12], the throughput improvements of pipelining are degraded by the data dependencies between interval subdivision and renormalization as they are implemented in separate stages. This results in an overall throughput of 0.59 bins/cycle since pipeline stalls are necessary. To mitigate this issue, the pipeline division in our implementation is made between *range* and *low* updates instead, with both interval subdivision and renormalization pertaining to each variable implemented within the same cycle. This means that no stalls are issued due to data dependencies across pipeline stages.

The context memory in this implementation can be considered an extreme case of the multiple SRAM bank architecture proposed in [18]. Since the more important design criteria is throughput and area is of secondary concern in the H.264 encoder as discussed above, a register bank structure is used so that no stalls are necessary for context access.

5.4.2 Critical Path Reduction Versus Area Overhead

The area of the different modular components in the design are summarized in Table 5.3. These area numbers correspond to the original critical path length. With the optimized version synthesized, the gate count of the *range* update pipeline stage increases from 6.43K to 8.81K, but the timing path is reduced to 20.05 ns. The increase in area is acceptable since the overall area of the entropy encoder is dominated by the context memory and its lookup pipeline stage, as shown in Table 5.3.

Table 5.3: Modular area distribution in H.264 entropy encoder design.

Module	**Gate Count**
Context Lookup and Update	334.7K
range update with renormalization	6.43K
low update with renormalization	78.94K
Total	452.6K

Chapter 6

Conclusions

The H.264 and VC-1 entropy encoding schemes provide a stark contrast in terms of implementation, and as a result different engineering trade-offs were made for the two entropy encoding units. In the VLC architecture, timing is not as important because there are less dependencies between concurrently encoded syntax elements. Thus, optimization techniques mainly targeted minimizing the amount of area overhead incurred due to parallelism. Conversely, the CABAC architecture needs to be designed and optimized to meet timing constraints, as the serial nature of the operations prevents bins to be encoded in parallel, so each additional bin encoded in a cycle increases the critical path length. For this reason a pipeline architecture is used and area overhead from pre-computing values for multiple candidates is deemed acceptable for the savings in logic delay.

The VC-1 entropy encoder was synthesized into a total gate count of 136.6K and total power of 304.5 μW. The corresponding critical path delay is 6.5 ns. The number of syntax elements encoded per second is 735M. Two major components of the encoder architecture were targeted for area overhead reduction – variable length code lookup tables and bit concatenation logic. The use of parallelism necessarily requires duplication of hard-coded lookup tables which results in area and power costs. Sharing of low-usage tables that are guaranteed to be accessed by at most one of the three parallel *VLC* engines reduces the area and power overhead significantly; tables with large discrepancies in code length are partitioned and low-probability (long) entries are shared between the three engines to further reduce area and power. A hierarchical structure is chosen over a single-level bit concatenation scheme do to savings in bitwidth and as a result area and power consumed

by large barrel shifters that concatenate consecutive codewords into a single bitstream.

The H.264 entropy encoder was synthesized and further optimizations were implemented to reduce the critical path delay from 27.38 ns to 20.05 ns, while suffering an increased gate count from 411.3K to 452.6K. A pipelined architecture is chosen to allow multiple binary symbols to be encoded in a single cycle, which is a specification rendered critical by the throughput requirements of high-resolution real-time encoding. Serial dependencies result in significant increase in the critical timing path with each additional symbol encoded per cycle, so a form of precomputation is used to meet timing constraints. A register file implementation and use of bypass datapaths are also exploited to eliminate the need for pipeline stalls during context memory accesses. Due to three-frame parallelism and the cascaded processing of 6 bins in a pipelined architecture, the encoder can achieve a bin-rate of 4.97G binary symbols per second, which far exceeds those reported for previous works.

6.1 Future Work

To extend the functionality of the entropy encoder module for portable video encoding applications, the binarization and context selection portions can be investigated and incorporated into the design. The impact of the added complexity may be justified if there are possible optimizations across the binarization and core CABAC encoding function. It may be constructive to examine the similarities between the variable length coding (VLC) portion of the binarization step in H.264 entropy encoding and the core functionality of the VLC-based entropy encoding block in VC-1.

The bitpacking functionality can also be optimized to be shared between the VC-1 and H.264 entropy encoding modules to a greater extent. Since only one engine is enabled at once, some efficient reuse scheme can be employed to reduce the overall area consumption of the bitpacking unit.

With the increasing demand for reconfigurability in multimedia systems, the current design may be extended to support a wide range of video standards by identifying the similarities in core functionality and abstracting the syntax-specific information from the entropy encoder functional block. It may then be possible to implement a truly reconfigurable system that utilizes two distinct architectures for variable length coding and arithmetic coding based video standards, with peripheral logic handling the mapping from standard-specific

syntax elements to a generic set of inputs to the core.

Appendix A

Nomenclature

A.1 Common Terminology [1, 3]

B slice Bi-predictive slice – can be decoded using intra-prediction or inter-prediction with at most two motion vectors and reference indices to predict the sample values of the block.

block An array of samples or coefficients.

bitstream A sequence of bits generated by the entropy encoder to represent the video data.

coefficient A scalar quantity in the frequency domain.

chroma Representing one of two colour difference signals related to the primary colours (Cb and Cr).

FIFO A first-in, first-out buffer.

frame	A single picture in a video sequence; consists of an array of *luma* samples and two arrays of *chroma* samples.
I slice	A slice that is decoded using intra-prediction only.
luma	Representing the monochrome signal related to the primary colours.
macroblock	A 16x16 block of luma samples and two corresponding blocks of chroma samples of a picture.
motion vector	A two-dimensional vector used for inter-reprediction that provides an offset from the coordinates in the current picture to coordinates in a reference picture.
P slice	A slice that can be decoded using intra-prediction or inter-prediction with at most two motion vectors and reference indices to predict the sample values of the block.
residual	The decoded difference between a prediction of a sample or data element and its decoded value.
syntax element	A unit of data presented in the *bitstream.*
ultra-HD	Display format with 4096x2160 pixels per frame (also known as 4Kx2K, quad-full-HD); number of pixels is equivalent to four times that of a full-HD (1080p, i.e. 1920x1080 pixels per frame).

A.2 VC-1 (VLC) Terminology [3]

entry-point layer A point in the bitstream that offers random access.

FLC Fixed length code or fixed length coding; category of syntax elements defined in VC-1 that do not require a table lookup.

picture layer Equivalent to a frame containing lines of spatial information of a video signal.

sequence layer A coded representation of a series of one or more pictures.

VLC Variable length code or variable length coding; lookup-table-based entropy encoding defined by VC-1 Standard.

A.3 H.264 (CABAC) Terminology [1]

bins	Binary representation of syntax elements.
binarization	A unique mapping process of all possible values of a syntax element into bins.
CABAC	Context-Based Adaptive Binary Arithmetic Coding.
context	Probability model.
picture parameter set	A syntax structure containing syntax elements that apply to zero or more entire coded pictures as determined by the pic_parameter_set_id syntax element found in each slice header.
sequence parameter set	A syntax structure containing syntax elements that apply to zero or more entire coded video sequences as determined by the seq_parameter_set_id syntax element found in the picture parameter set.
slice header	A part of a coded slice containing the data elements pertaining to the first or all macroblocks in a slice.

Bibliography

[1] *Recommendation ITU-T H.264: Advanced Video Coding for Generic Audiovisual Services*, ITU-T Rec. H.264 Std., 2009.

[2] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low power cmos digital design," 1995. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.9965

[3] *SMPTE 421M-2006, VC-1 Compressed Video Bitstream Format and Decoding Process*, The Society of Motion Picture and Television Engineers Std., 2006.

[4] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1011, Sept. 1952.

[5] S.-M. Lei and M.-T. Sun, "An entropy coding system for digital hdtv applications," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 1, no. 1, pp. 147 –155, march 1991.

[6] J.-C. Chu, L.-F. Su, Y.-C. Yang, J.-I. Guo, and C.-L. Su, "A multi-mode entropy decoder with a generic table partition strategy," in *SoC Design Conference, 2008. ISOCC '08. International*, vol. 01, 24-25 2008, pp. I–25 –I–28.

[7] Y.-H. Lim, S.-S. Jun, and J.-S. Kang, "An efficient architecture of bitplane coding in vc-1 for real-time video processing," in *Signal Processing and Information Technology, 2007 IEEE International Symposium on*, 15-18 2007, pp. 1198 –1203.

[8] S. Cho, "A low power variable length decoder for mpeg2 decoder system," Master's thesis, Massachusetts Institute of Technology, May 1997.

[9] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, June 1987.

[10] P. G. Howard and J. S. Vitter, "Arithmetic coding for data compression," *Proceedings of the IEEE*, vol. 82, no. 6, pp. 857–865, June 1994.

[11] R. Osorio and J. Bruguera, "A new architecture for fast arithmetic coding in h.264 advanced video coder," in *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on*, 30 2005, pp. 298 – 305.

[12] L. Li, Y. Song, T. Ikenaga, and S. Goto, "A cabac encoding core with dynamic pipeline for h.264/avc main profile," *ASPCAC2006*, pp. 761–764, Dec 2006.

[13] C.-C. Kuo and S.-F. Lei, "Design of a low power architecture for cabac encoder in h.264," in *Proceedings of IEEE Asia Pacific Conference on Circuits and Systems*, 2006, pp. 243–246.

[14] O. Flordal, D. Wu, and D. Liu, "Accelerating cabac encoding for multistandard media with configurability," in *Proceedings of 20th International Parallel and Distributed Processing Symposium*, 2006.

[15] J.-L. Chen, Y.-K. Lin, , and T.-S. Chang, "A low cost context adaptive arithmetic coder for h.264/mpeg-4 avc video coding," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, 2007, pp. II–105–108.

[16] P.-S. Liu, J.-W. Chen, and Y.-L. Lin, "A hardwired context-based adaptive binary arithmetic encoder for h. 264 advanced video coding," in *Proceedings of International Symposium on VLSI Design, Automation and Test*, 2007, pp. 1–4.

[17] C.-C. Lo, Y.-J. Zeng, and M.-D. Shieh, "Design and test of a highthroughput cabac encoder," in *Proceedings of IEEE Region 10 Conference*, 2007, pp. 1–4.

[18] Y.-J. Chen, C.-H. Tsai, and L.-G. Chen, "Architecture design of area-efficient sram-based multi-symbol arithmetic encoder in h.264/avc," in *Proceedings of IEEE International Symposium on Circuits and Systems*, 2006, pp. 1–4.

[19] J. Nunez-Yanez, V. Chouliaras, D. Alfonso, and F. Rovati, "Hardware assisted rate distortion optimization with embedded cabac accelerator for the h.264 advanced video codec," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 2, pp. 590–597, 2006.

[20] R. Osorio and J. Bruguera, "Arithmetic coding architecture for h.264/avc cabac compression system," in *Proceedings of Euromicro Symposium on Digital System Design*, 2004, pp. 62–69.

[21] ——, "High-throughput architecture for h.264/avc cabac compression system," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 16, no. 11, pp. 1376–1384, 2006.

[22] X. Tian, T. Le, X. Jiang, and Y. Lian, "A hw cabac encoder with efficient context access scheme for h.264/avc," in *Proceedings of IEEE International Symposium on Circuits and Systems*, 2008, pp. 37–40.

[23] H. Shojania and S. Sudharsanan, "A high performance cabac encoder," in *IEEE-NEWCAS Conference, 2005. The 3rd International*, 19-22 2005, pp. 315 – 318.